MASTER EN SCIENCES INFORMATIQUES

Simula
extension de la classe simulation

Van Impe, Thierry

Award date:
1979

Awarding institution:
Universite de Namur

FACULTÉS UNIVERSITAIRES
NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

S I M U L A

EXTENSION DE LA CLASSE SIMULATION

TH. VAN IMPE

Je remercie Monsieur le Professeur Jean Fichefet pour son assistance et ses conseils nécessaires à l'élaboration de ce mémoire.

Je tiens également à remercier le Professeur Madame Monique Noirhomme-Fraiture pour l'aide prodiguée tout au long de ce travail.

Je remercie aussi le Professeur Paul Bratley de l'Université de Montréal ainsi que son équipe pour les conseils utiles reçus pour la rédaction de mon mémoire ainsi que pour la bonne intégration au sein du département.

°

° °

TABLE DES MATIERES

page

page

°

° °

CHAPITRE I

La Simulation

L'idée de simuler est aussi ancienne que l'homme a été une des premières manifestations de l'intelligence de notre espèce.

Les Sumériens faisaient des modèles de leurs canaux d'irrigation - les anciens Egyptiens faisaient des modèles de leurs projets de pyramides, l'idée de maquette et de modèle réduit peut être retrouvée dans tous les travaux des ingénieurs civils et militaires au cours de toute l'histoire connue.

Une pièce de théâtre est une simulation d'une suite de simulations, un match de football ou de rugby est une simulation de bataille selon des règles très particulières, toutes ces simulations sont éducatives et simulent l'intelligence face aux situations complexes du réel. De telles simulations sont donc réalisées d'une façon artificielle par rapport au réel observé et analysé; elles peuvent se rapporter à des phénomènes physiques, phénomènes sociaux et économiques.

La simulation est donc une construction artificielle, destinée à imiter des phénomènes naturels, avec une telle exactitude qu'on puisse confondre le réel et son modèle.
Avec l'ordinateur, le concept de simulation a pris un aspect nouveau qui en multiplie les applications et la facilité.
Le phénomène à étudier peut être répété à partir des instructions fournies mais à des vitesses beaucoup plus grandes que dans la réalité. Tout ce qu'un ordinateur fait, un groupe d'hommes peut le faire, même les simulations les plus compliquées, mais la lenteur de cette "machine humaine" la rend impropre.

L'ordinateur permet donc de faire des expériences artificielles sur toutes sortes de phénomènes économiques et sociaux. Files d'attente, gestion de stocks, investissements, processus de distribution, compactement, marketing, etc.

La taille des phénomènes à simuler ne fait plus reculer les utilisateurs d'ordinateurs : gestion d'un port de commerce, un réseau urbain, une usine, un réseau interconnecté de liaisons de communications, un aéroport sont couramment soumis à des simulations. Elles apportent des informations précieuses sur les performances des systèmes, elles permettent de comparer des politiques, voire des stratégies, elles donnent donc des possibilités d'expérimentation presque illimitées dans de nombreux domaines.

On pourrait encore définir la simulation comme étant une technique qui utilise l'expérimentation sur des modèles pour étudier le comportement de systèmes complexes.
Précisons, dans ce qui suit, la notion de système et de modèle.

Un système est peut être considéré comme une collection d'objets ayant certaines *caractéristiques* qui prennent part à des activités. Les objets seront aussi appelés "entités" et les caractéristiques "attributs".
La description exacte de la valeur de tous les attributs et des actions en cours à un moment donné est appelée *état du système*.
Pour classifier les différents systèmes, nous dirons qu'un système est *dynamique* lorsque les attributs varient avec le temps tandis qu'un système est *statique* lorsque le temps ne joue aucun rôle.
Dans un système dynamique, si après initialisation, les attributs varient de manière continue dans le temps, le système est dit *continu*; autrement le système est *discret*. Un changement discret à l'état du système est appelé *événement*.
Certaines classes de systèmes se prêtent à une analyse mathématique tandis que d'autres se prêtent mieux à la simulation. Les systèmes simulés sont les plus souvent dynamiques et discrets.

Un modèle est une représentation d'un système et est toujours une approximation qu'on voudrait rendre aussi fidèle à la réalité que possible.
Le choix d'un modèle approprié est chose délicate car il faut dégager les aspects essentiels du système et ne pas prendre en considération trop de détails; ce qui entraînerait un alourdissement du modèle.
La simulation n'est pas la seule technique possible pour l'étude de systèmes. D'autres techniques plus classiques, l'observation directe et l'analyse mathématique lui sont dans certains cas préférables.
La simulation coûte cher, elle devrait dès lors être une technique de dernier recours, à employer si aucune autre est utilisable.

Il existe deux types de langages de simulation : ceux pour les systèmes discrets et ceux pour les systèmes continus. Nous en détaillerons que ceux pour les langages discrets.

Le problème classique de la simulation discrète est celui des files d'attente. Des entités provoquent des actions demandant l'utilisation de ressources partageables. Quand les ressources ne sont pas disponibles, des files d'attente se créent.
Il est habituel de différencier les entités physiques du système en deux catégories. Les entités *actives* appelées transactions ou processus, se déplacent dans le système utilisant les services des entités passives ou ressources. Le plus souvent, le nombre des entités actives varie durant la simulation tandis que les ressources restent fixes.
Comme exemple, on pourrait nommer Parking, un système qui comprend des entités actives; les véhicules voulant stationner et les entités passives; les emplacements disponibles. La file d'attente existera au moment où tous les emplacements sont occupés. Le système est bien discret car l'état du système ou la longueur de la file d'attente ou le degré d'occupation de la resource, change seulement quand une transaction prend possession d'une ressource, la libère, rentre dans la file, sort du système ...

Nous verrons que l'élément moteur d'une simulation discrète est *l'échéancier* dans lequel sont placées des notices d'événements indiquant l'heure et le type des événements prévus.

Dans ce rapport, deux langages de simulation discrète sont développés : GPSS et SIMULA.
Le langage GPSS est surtout destiné à la simulation des systèmes de files d'attente. Le concept de transaction existe pour modeler les entités actives, les concepts de"facility" et "storage" pour représenter des ressources unitaires et divisibles respectivement.
Les files sont gérées automatiquement.

A l'encontre de GPSS qui est un langage hautement spécialisé, SIMULA est un langage de programmation générale qui se place au même niveau que PL/1, PASCAL ou ALGOL 68.
Le langage est extensible et c'est une extension standard qui en fait un langage de simulation.

Le but de cet ouvrage est d'offrir à cette extension de SIMULA, les avantages que nous offrait GPSS car SIMULA est dépourvu de toute gestion de files d'attente et de concepts de ressources.
C'est pourquoi, notre démarche sera d'étudier les langages GPSS et SIMULA avant d'approfondir une nouvelle extension à SIMULA qui réalise cette quasi-similitude d'efficacité que le langage GPSS.

Enfin, de nouvelles procédures ont été implémentées dans le but d'étendre les possibilités offertes au programmeur pour une analyse judicieuse des résultats de simulation.

o

o o

CHAPITRE II

LE LANGAGE G P S S

2.0. INTRODUCTION

GPSS (General Purpose Simulation System) est un des plus vieux langages de simulation discrète. Il est aussi le plus répandu.
Introduit en 1961 par G. GORDON d'I.B.M., pour l'ordinateur IBM 7090, il a vu une amélioration continue à travers plusieurs versions. Le système fonctionne sur les ordinateurs de Xérox, Control Data, Univac, Dec, Siemens et bien d'autres.

2.1. BLOCS ET TRANSACTIONS

Un modèle en GPSS prend la forme d'un organigramme. Une quarantaine de blocs différents sont disponibles, avec des formes et fonctions prédéfinies. La construction d'un modèle est équivalente à la sélection et connexion d'un certain nombre de blocs.

Les entités mobiles et actives d'un modèle GPSS s'appellent les TRANSACTIONS. Au début de l'exécution d'un modèle, aucune transaction n'existe. A mesure que la simulation se déroule, des transactions sont créées à différents points de l'organigramme. Une fois qu'une transaction a été créée, elle peut passer d'un bloc à l'autre d'un modèle suivant les flèches qui relient les différents blocs.

Certains blocs vont retenir la transaction pour une période de temps simulée; d'autres peuvent détruire la transaction. L'entrée d'une transaction dans certains blocs provoquera un changement d'état dans le modèle; et finalement certains blocs peuvent refuser d'admettre une transaction, qui doit donc attendre ou aller ailleurs.

Exemple

```
GENERATE 10,5
SEIZE CPU
ADVANCE 8,4
RELEASE CPU
TERMINATE 1
```

2.2 L'HORLOGE

GPSS tient à jour une horloge simulée. Cette horloge ne prend que des valeurs entières. A chaque transaction, on attribue ce qu'on appelle un BDT (temps de départ du bloc ou Bloc Departure Time).

Le système constitue ainsi une liste des BDT et ne commence qu'à l'instant où une transaction est susceptible d'avancer dans le système. Les transactions dans un modèle GPSS sont sur l'une ou l'autre des deux listes suivantes :

- la chaîne actuelle ou liste des événements courants (LEC) qui accueille les transactions dont le BDT est inférieur ou égal à l'instant t de l'horloge;
- la chaîne future ou liste des événements futurs (LEF) qui accueille les transactions dont le BDT est supérieur à l'instant t de l'horloge.

Prenons un exemple :

```
SIMULATE
GENERATE   15   Arrivée des voitures toutes les 15 sec.
QUEUE      1    La voiture fait la queue.
SEIZE      1    La voiture entre dans le péage.
DEPART     1    Elle quitte donc la queue.
ADVANCE    20   La voiture occupe le péage pendant 20 sec.
RELEASE    1    La voiture libère le péage
```

La première transaction à l'instant 15 va rentrer dans les blocs QUEUE, SEIZE et DEPART sans rencontrer aucun obstacle. Dans le bloc ADVANCE son BDT va être mis à 35 (20 + 15). Cette transaction est donc placée dans la LEF, la seconde transaction ne va pas pouvoir saisir la station n° 1 (SEIZE) puisque la première transaction l'occupe. Elle restera dans une file d'attente interne de la station n° 1 (SEIZE) avec un statut d'attente. Ce n'est que quand la transaction n° 1 libèrera la station simple n° 1 que la deuxième transaction recevra automatiquement un statut actif et sera placée dans la LEC à ce temps t.

Donc, une chaîne interne est associée à certains blocs (SEIZE, ENTER...) qui contient les transactions voulant saisir cette station alors qu'elle serait occupée. Ces transactions provenant de la LEC auraient, dans cette chaîne d'attente, un statut passif car leur BDT serait inférieur au temps t.

Durant le déroulement d'une simulation, GPSS commence par parcourir toutes les transactions qui se trouvent sur la LEC. D'abord la première transaction de cette liste est avancée aussi loin que possible dans l'organigramme. Elle passe de bloc en bloc (en provoquant éventuellement des actions sur l'état du modèle) jusqu'au moment où elle est arrêtée par une des trois raisons suivantes :

- Elle passe dans un bloc qui doit la retenir pour une période de temps simulée. Dès lors, la transaction est enlevée de la LEC pour être envoyée dans la LEF avec une indication de temps où elle sera libre de continuer.
- Elle passe dans un bloc qui la détruit.
- Le bloc suivant dans son chemin refuse de la laisser rentrer. Dans ce cas, la transaction reste dans la LEC, mais son progrès est terminé pour le moment.

Quand la première transaction de la LEC est arrêtée, la deuxième est ensuite avancée dans l'organigramme, et ainsi de suite pour les autres transactions de la LEC.

Quand la LEC a été complètement parcourue, on cherche dans la LEF la ou les transactions avec le plus petit BDT. Les transactions en question sont enlevées de la LEF et mises dans la LEC, l'horloge simulée est avancée à la valeur indiquée et on recommence à parcourir la LEC.

2.3. CREATION ET DESTRUCTION DES TRANSACTIONS

Le bloc GENERATE permet de créer des transactions. L'écriture de tout bloc de GPSS (à l'exception du bloc BUFFER) comprend plusieurs champs possibles, le premier champ du bloc GENERATE décrit le temps moyen qui sépare la génération de deux transactions.

Par exemple, GENERATE 10 aura pour effet de créer normalement une transaction aux temps t=10, t=20, t=30, ...

Le deuxième champ nous donne un modificateur du temps moyen. Ce modificateur peut être uniforme, représenté par une fonction prédéfinie...
Par exemple, GENERATE 10,5 signifie que le temps entre la génération de deux transactions sera une valeur aléatoire entière prise dans l'intervalle (5,15) avec probabilité égale.

Le bloc TERMINATE détruit les transactions qui y arrivent. A chaque simulation est associé un compteur de terminaison (CT). Si une transaction arrive dans un bloc TERMINATE qui a un champ non-zéro, la valeur de ce champ sera déduite du CT, par exemple chaque fois qu'une transaction arrive dans le bloc TERMINATE 2, la transaction sera détruite et on soustrait 2 de la valeur du CT. Quand la valeur du CT devient zéro (ou négatif), la simulation est arrêtée.

La valeur initiale du CT est déterminée par une carte de contrôle START.

Exemple :

```
GENERATE 1000   Au moment t=1000, une transaction quittera le bloc
TERMINATE  1    GENERATE et sera tout de suite détruite par le bloc
START      1    TERMINATE. Le CT qui était initialisé à 1 deviendra
                0 et la simulation sera arrêtée, quel que soit l'état
                du modèle.
```

Le bloc ADVANCE retarde les transactions dans le système pendant un nombre d'unités de temps égal au temps spécifié dans les différents champs de ce bloc. C'est ainsi que le bloc ADVANCE sera utilisé dans les stations simples ou multiples pour y simuler le temps que les transactions y passent en réalité. Par exemple, ADVANCE 60 aura pour effet d'enlever la transaction de la LEC et d'ajouter 60 à son BDT avant de l'envoyer dans la LEF.

Le bloc GENERATE n'est pas le seul à créer des transactions. Il en existe un autre, le bloc SPLIT qui fabrique des copies de la transactions actuelle. La transaction initiale et ses copies forment alors une famille de transactions :

```
SPLIT  nb de copies , bloc suivant pour les copies
          /                    /
       Champ A              Champ B
```

Le bloc ASSEMBLE récupère les transactions d'une même famille; celle qui arrivera la première au bloc ASSEMBLE pourra repartir, elle seule, vers les autres blocs.

Le bloc GATHER a un rôle voisin de celui du bloc ASSEMBLE. La différence est qu'aucune transaction n'est détruite par le bloc GATHER.

2.4. UNITES DE TRAITEMENT ET RUPTURES DE SEQUENCE

2.4.1. Les stations simples

Elles ne peuvent traiter qu'une seule transaction à la fois. Cette transaction peut entrer de deux façons différentes dans une station simple ou ressource.

La première façon, qui est la plus courante, est l'emploi du bloc SEIZE.

La deuxième façon de pénétrer dans la station simple est le bloc PREEMPT.

Si la station simple est déjà occupée par une autre transaction, celle voulant y pénétrer sera placée dans une chaîne d'attente associée à la station, jusqu'au moment où cette ressource sera à nouveau libérée.

Après avoir passé un certain temps dans la ressource (à l'aide d'un bloc ADVANCE par exemple), il faudra que la transaction la libère. Elle le fera à l'aide du bloc RELEASE.

A partir du moment où une ressource devient libre, une autre transaction peut la saisir. Si plusieurs transactions attendent la libération de la même station, les demandes sont normalement satisfaites sur base FIFO (first in, first out).

Le bloc PREEMPT permet à certaines transactions de saisir la station simple de façon prioritaire. Son bloc associé est le RETURN tout comme le bloc SEIZE avait un bloc RELEASE qui lui était associé.

2.4.2. Les blocs TRANSFER LOOP MATCH

Le bloc TRANSFER permet de diriger des transactions vers un bloc, ou des blocs, qui dans le programme n'est ou ne sont pas directement à la suite du bloc TRANSFER.

Le bloc LOOP permet de faire passer plusieurs fois les transactions dans une même partie du programme. L'intérêt de ce bloc est qu'il peut contrôler le nombre de fois qu'une transaction est passée dans une section donnée de l'organigramme.

Le bloc MATCH est un moyen de coordonner la marche de deux transactions dans le modèle, c'est-à-dire de les synchroniser, sans les obliger à se retrouver au même endroit. On utilise alors ce bloc MATCH, pour synchroniser deux flots de transactions de la même famille.

2.4.3. Les files d'attente

Si on désire recevoir des statistiques sur les files d'attente, il faut incorporer dans son modèle des entités appelées Queues.

Il est important de souligner qu'une queue n'est nécessaire que pour la collecte de statistiques : les files d'attente internes de GPSS nécessaires pour la bonne gestion des ressources ou stations sont toujours implicitement présentes, et la bonne marche du modèle ne dépend pas de la présence ou absence de queues.

Par exemple, une transaction devient membre d'une queue LINE quand elle entre dans le bloc QUEUE LINE. Pour qu'elle devienne non-membre, il lui faudra passer par le bloc DEPART LINE.

Quand une transaction devient membre d'une queue, on note dans la transaction la valeur de l'horloge simulée. De même, quand elle quitte la queue, l'horloge est lue à nouveau. De cette façon, outre les statistiques concernant

la longueur d'une queue, GPSS peut tenir à jour des statistiques sur le temps simulé que chaque transaction a passé dans la queue. Toutes ces statistiques sont imprimées automatiquement à la fin de la simulation.

2.4.4. Les stations multiples

Les stations multiples représentent des unités de traitement qui ne peuvent traiter qu'un nombre limité de transactions, simultanément.
Ce nombre caractéristique de la station multiple s'appelle capacité.

Le bloc ENTER a pour bloc associé le bloc LEAVE.

ENTER n° station multiple , nb d'unités qui entrent dans la station multiple.
LEAVE n° station multiple , nb d'unités qui sortent de la station multiple.

2.5. CONTROLE LOGIQUE DU FLUX DES TRANSACTIONS

Toute transaction créée par un bloc GENERATE comporte automatiquement 12 paramètres d'un demi-mot chacun (un champ du bloc GENERATE permet de faire varier de 0 à 100 ce nombre de paramètres). Le bloc ASSIGN est le principal moyen d'attribuer des valeurs numériques entières à des paramètres donnés des transactions.

Le bloc TEST a pour but de contrôler le passage des transactions, soit dans le bloc suivant, soit dans un autre bloc, à l'aide d'opérateurs de comparaison : TEST (opérateur) champ A, champ B, champ C). L'opérateur de comparaison est du genre L LE E NE G GE.
Le champ C désigne une étiquette de bloc vers laquelle la transaction se dirigera si la relation entre le champ A et le champ B n'est pas réalisée. Par contre, si elle est réalisée, la transaction se dirigera vers le bloc immédiatement à la suite du bloc TEST.

Le bloc LOGIC permet de positionner ou d'inverser un interrupteur logique défini : LOGIC $\begin{matrix}R\\S\\I\end{matrix}$ n° d'identification de l'interrupteur R = reset, S = set, I = inverse.

Le bloc JOIN permet de définir des groupes. En effet, les transactions du groupe K, par exemple, seront les transactions qui seront passées dans un bloc JOIN ayant le chiffre K comme champ du bloc.

Le bloc REMOVE permet d'exclure une transaction qui faisait partie du groupe.

Le bloc EXAMINE permet de tester si une transaction appartient ou non au groupe K, par exemple.

Le bloc GATE, comme le bloc TEST, contrôle le flux des transactions. Mais alors que dans le bloc TEST on utilise des comparaisons entre des nombres, on utilise ici des comparaisons entre différents attributs ou caractéristiques numériques que peuvent posséder les différentes entités de GPSS (ex: nombre de transactions dans la file d'attente n° j; nombre de transactions ayant pénétré dans un bloc; temps courant du système, ...).

2.6. CONTROLE DE L'EXPLOITATION ET DE L'OUTPUT

A l'issue de la simulation d'un modèle, GPSS édite des statistiques. L'utilisateur peut désirer des informations plus détaillées que celles fournies automatiquement. Les tables de distribution par l'intermédiaire du bloc TABULATE et quelquefois du bloc MARK donnent la répartition de n'importe quel attribut au cours de la simulation.

Quand une transaction est créée par un blanc GENERATE, l'heure absolue du système à cet instant est enregistrée dans un mot attaché à la transaction : le mot MARK TIME.
Le bloc MARK permet de modifier le MARK TIME d'une transaction et donc de mesurer d'autres termes de transit. Quand une transaction passe dans un bloc MARK, l'heure absolue à cet instant remplace l'heure de création de la transaction.

GPSS possède également les tables de distribution.

Un bloc TABULATE fait référence à une déclaration de TABLE (où on définit le caractère dont on veut la distribution, on donne aussi la limite supérieure du premier intervalle et la longueur des intervalles) qui enregistre les valeurs du caractère donné. Lorsqu'une transaction entre dans le bloc TABULATE, la valeur de ce caractère est enregistrée par la table correspondante.

Le bloc SAVEVALUE permet de conserver en mémoire la valeur de certains attributs en variable globale du programme, afin par exemple d'utiliser cette valeur déterminée dans la suite de la simulation, l'attribut entre-temps ayant pu varier.

Ex: SAVEVALUE 10, P1 : la valeur du paramètre 1 de la transaction passant dans ce bloc SAVEVALUE sera enregistrée dans une zone mémoire n° 10.

°

° °

CHAPITRE III

LE LANGAGE SIMULA

3.0. INTRODUCTION

SIMULA 67 est un langage évolué, créé au centre de calcul norvégien par O.J. Dahl et K. Nygaard avec la collaboration de B. Myhraug.
Le langage prend ALGOL comme base de départ et lui ajoute certains concepts pour rendre facile le traitement de données structurées et la simulation. Certains points d'ALGOL, comme les entrées-sorties et le traitement de chaîne de caractères, ont été complètement repensés.

La sémantique du langage a aussi été légèrement modifiée pour permettre plus de vérifications à la compilation et rendre l'exécution plus rapide. Comme son nom l'indique, SIMULA a été conçu pour la simulation, mais la puissance du langage permet de traiter une très grande classe d'applications.

En gros, il y a trois aspects du langage qui le caractérisent :

1) Existence de pointeurs (variable "REF") et l'allocation de blocs de mémoire sous contrôle du programmeur. Ceci permet le traitement de liste. Le ramassage de miettes (garbage collection) se fait automatiquement.
2) Le concept de "CLASS" qui comprend celui des based-structures de PL/1 et celui des coroutines.
3) Un mécanisme de déclaration hiérarchique qui donne au langage une possibilité d'extension.

Le langage SIMULA contient déjà deux classes de système précompilées, SIMSET et SIMULATION, qui ajoutent au langage de base les structures et procédures utiles au traitement de listes et à la simulation.
Le programmeur peut se servir du même mécanisme pour créer des classes utiles à certaines catégories d'applications.

3.1. LA BASE ALGORITHMIQUE ET SYNTAXIQUE

Dans la description qui suit, nous écrirons en minuscules et soulignerons les mots-clés de SIMULA.

L'élément de base d'un programme ALGOL est le BLOC. Il est défini comme suit :

begin - déclarations suivies chacune d'un ; -
- énoncés séparés par un ; -

end ;

En SIMULA, un bloc peut être préfixé par un nom de CLASSE (voir cette section des notes).
En ALGOL et en SIMULA, le programme lui-même est un bloc.

Un bloc se comporte comme un appel de procédure sans nom ni paramètre . Les déclarations à l'intérieur du bloc définissent donc des identificateurs locaux à ce bloc et l'allocation de la mémoire pour les variables ainsi définies se fait au moment de l'entrée dans le bloc (on a donc une allocation dynamique de la mémoire).

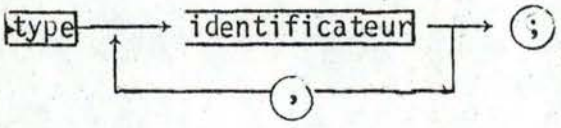
3.1.1. Déclarations et constantes

En SIMULA, on a sept types de base : integer, real, character, boolean ref(xxx), text et switch.
Il y a une constante de type ref, c'est none (où xxx représente alors le pointeur vide).
En SIMULA, il y a une distinction entre les caractères et les chaînes de caractères (text). Les constantes de type character sont écrites entre ' et '.
Les chaînes sont écrites entre deux paires de double apostrophes ("").
Exemple : 'A'
""CECI EST UNE CHAINE""
""A"" (chaîne formée d'un seul caractère).

Il faut noter qu'excepté dans les chaînes et dans ' x ' , les blancs (espace) sont non-significatifs, ils sont supprimés dans une des premières étapes de la compilation.
Exemple : 1 5 7 est la constante entière 157
TEMPS D'ATTENTE est un identificateur acceptable.

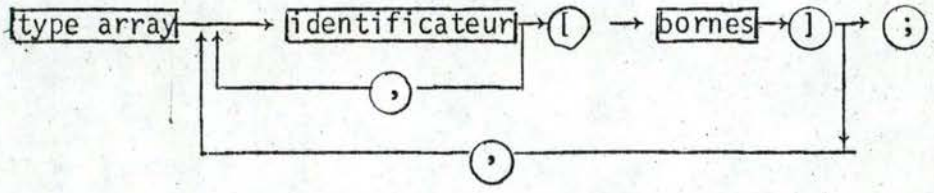
On a les diagrammes syntaxiques suivants :

Déclaration de variables simples

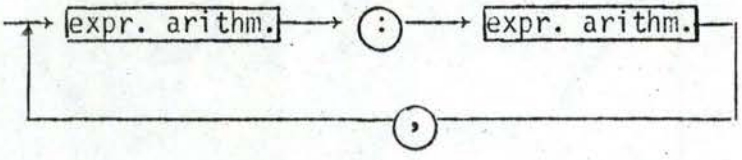


Ex: real x;
integer I,J,K;

Déclaration de tableaux



Bornes :



Ex: integer array AGE [I : 50];
real array X,Y [I : 10,1 : 15] , Z [9 : 15] ;

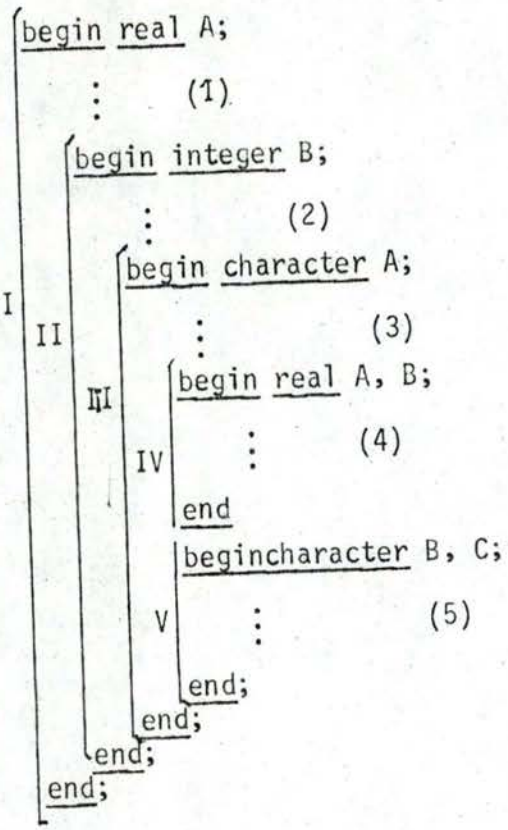
Au moment de l'allocation de la mémoire pour les variables déclarées dans un bloc, ces variables sont initialisées à des valeurs nulles :

pour les entiers et réels : 0.
pour les booléens : false.
pour les caractères : blanc. (compilateur NRDE)
pour les chaînes : notext.
pour les pointeurs (ref) : none.

3.1.2. Portée des identificateurs

La structure de blocs gouverne la portée des identificateurs. On ne peut utiliser un identificateur à l'extérieur du bloc dans lequel il a été défini;

on peut redéfinir un identificateur dans un bloc intérieur à un autre bloc.
Prenons un exemple :



Au point (1), on peut utiliser seulement l'identificateur A défini au bloc I.

Au point (2), on peut utiliser les identificateurs A et B pour faire référence au réel A défini en I et à l'entier B défini en II.

Au point (3), on peut faire référence à l'entier B défini en II et au caractère A défini en III, mais on ne peut plus référer au réel A défini en I, car on a redéfini l'identificateur A.

Au point (4), on peut seulement accéder aux réels A et B déclarés en IV, car on a redéfini les deux identificateurs (le réel A est différent de celui déclaré en I).

Au point (5), on peut accéder aux caractères A, B et C déclarés en III et IV.

3.1.3. Les opérateurs

Opérateurs arithmétiques :

- + Addition.
- - Soustraction.
- × Multiplication.
- / Division réelle.
- ÷ Division entière.
- ↑ Puissance (A↑B A^B).

Opérateurs relationnels : ceux-ci dépendent en SIMULA du type des opérandes.

- Entiers, réels, caractères : $< \leqslant = \neq \geqslant >$
- Booléens : (equiv) est opérateur d'égalité
- Pointeurs ref(xxx) : == et =/= sont opérateurs d'égalité et d'inégalité.
- Chaînes : (cfr section sur les chaînes).

3.1.4. Enoncés de base

En SIMULA, il y a très peu d'énoncés de base. On a :

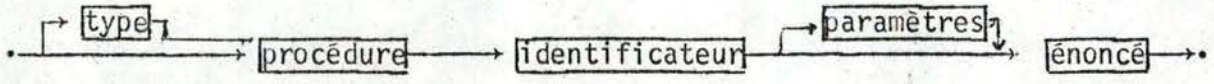
- des énoncés de type goto.
- des énoncés de type if.
- des énoncés de type for.
- des énoncés de connexion.
- des énoncés d'affectation.

L'opérateur d'affectation est := .

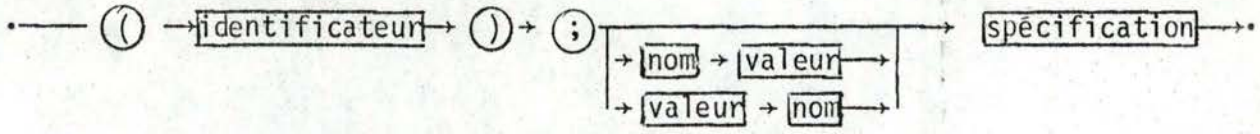
On peut faire des affectations multiples (A1:=A2:=A3:=A4...:=AN:=VALEUR;)
Les adresses sont évaluées de gauche à droite et les affectations se font de droite à gauche avec toutes les conversions nécessaires (ce qui fait que l'énoncé n'est pas équivalent dans tous les cas à la suite d'énoncés A1:=VALEUR; A2:=VALEUR; ... ; AN:=VALEUR;).
Pour les pointeurs, nous en reparlerons dans la section consacrée à la notion de CLASSE.

3.1.5. Procédures

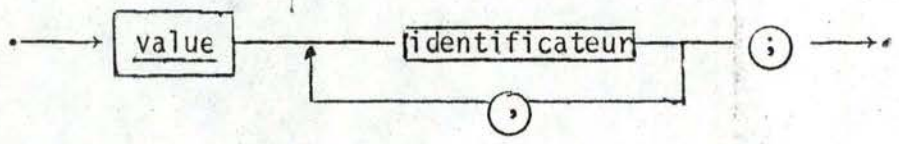
La syntaxe de la déclaration d'une procédure est la suivante :



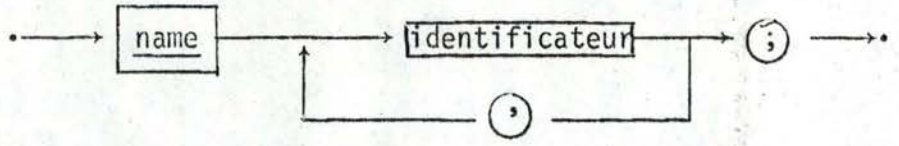
où paramètres est défini comme suit :



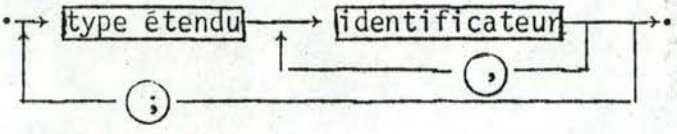
avec valeur :



avec nom :



avec spécification :



Quand un type apparaît devant une procédure, cela indique que l'on a une fonction.

Ex:
```
integer procedure MAX(A,B);
   value A,B;  integer A,B;
MAX := if  A < B   then B  else A;
```

Le type d'une procédure doit être simple. (integer,real,boolean, character, text,ref(xxx)).

Le type étendu pour les paramètres peut être un type simple ou :

1) -type- array
2) label
3) switch
4) procedure ou -type- procedure

Dans le cas d'un paramètre procedure, on ne spécifie pas ses paramètres et, dans le cas d'un tableau, on n'indique pas ses bornes.
Pour chaque type de paramètres, il y a un mode de passage qui peut être modifié par l'utilisation de name ou value.

Les modes de passage sont en SIMULA CDR 6000

	valeur	reference	nom
1) integer,real,boolean,character	D	X	O
2) ref(xxx)	X	D	O
3) text	O	D	O
4) tableaux de integer,real,boolean,character	O	D	O
5) tableaux de ref(xxx) et text	X	D	O
6) procedure	X	D	O
7) label	X	D	O
8) switch	X	D	O

X = interdit O = optionnel D = par défaut

Avec le compilateur de l'université de Montréal (NRDE), on ne peut utiliser de constante de textes quand on utilise l'option par défaut.
A quelques exceptions près, on passe les types simples par valeur et les types compliqués par référence.
Qnaud nous étudierons les classes, nous verrons que les modes de passage sont identiques avec les exceptions suivantes :

- on ne peut avoir de paramètres de type procedure, label ou switch;
- on ne peut passer de paramètres par nom.

3.2. ENTREES-SORTIES EN SIMULA

Bien que SIMULA soit basé sur ALGOL, il a ses propres procédures d'entrée et de sortie. Dans ces notes, nous ne traiterons que les E/S sur unités standards, c'est-à-dire lecteur de cartes et imprimante.

3.2.1. Généralités

Chaque fichier (entrée, sortie) possède un tampon. Un tampon est une image de carte (80 car) ou de ligne (136) en mémoire. Cette image constitue une chaîne de caractères qui sera traitée séquentiellement par les procédures d'entrée/sortie.

Un pointeur nommé "POS" indique le prochain caractère à traiter dans le tampon. Tous les échanges entre la mémoire et les fichiers externes se font à travers ces tampons.

Il est à noter que SIMULA ne traite que des fichiers de caractères, le mécanisme d'accès à des fichiers binaires n'étant pas défini.

3.2.2. Procédures d'entrée

Au début du programme, la première carte de données est transmise automatiquement au tampon d'entrée et le pointeur "POS" est placé au premier caractère. Pour changer de carte, on utilise la procédure INIMAGE.
INIMAGE entre dans le tampon la prochaine carte de données et fait pointer "POS" au premier caractère du tampon. S'il n'y a pas d'appel à INIMAGE, la prochaine lecture se fera sur l'image courante dans le tampon à partir de l'endroit où "POS" a été laissé.
Dans toutes les procédures d'entrée : si aucune donnée n'est trouvée quand "POS" atteint le dernier caractère du tampon, un INIMAGE est automatiquement fait et la recherche se continue sur la nouvelle image.

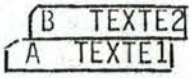
Lecture d'un caractère : character procedure INCHAR ;

INCHAR lit le caractère pointé par "POS" dans le tampon et augmente "POS" de 1. Ex: character A; A := INCHAR;

Lecture d'un entier : integer procedure ININT ;

ININT va chercher le prochain entier dans le tampon en sautant les blancs et "POS" s'arrête au premier blanc suivant cet entier (comme le format standard en ALGOL).

Ex: integer A; et comme image de carte

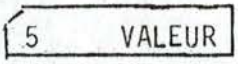


Pour lire A on fait : A := ININT;

Pour lire la lettre suivante, nous faisons à nouveau :

A := ININT; et ceci afin de sauter TEXTE1.

Dans un autre exemple, le programmeur contrôle lui-même le passage d'une carte à l'autre en se servant de INIMAGE. Cette technique serait utile si chaque carte avait un commentaire optionnel après l'entier :



Si on veut lire deux cartes ayant cette même image, on emploiera les quatre énoncés suivants pour la lecture de A et B :

```
A := ININT;
INIMAGE
B := ININT;
INIMAGE;
```

Lecture d'un réel : real procedure INREAL ;

INREAL cherche le prochain nombre dans le tampon, en sautant les blancs et "POS" s'arrête au premier blanc suivant ce nombre.

Ex: Si on a déclaré real B;

Pour lire, on fait B := INREAL;

Le nombre trouvé, réel ou entier, est représenté en flottant. Si un texte est trouvé à la place du nombre, il y aura erreur.

Lecture d'un texte : text procedure INTEXT(N) ;

INTEXT lit un texte de longueur N (N est un entier); N ne doit pas dépasser la longueur du tampon.

Les textes sont des variables qui doivent être initialisées avant d'y mettre une chaîne de caractères.

```
Ex:  text T1,T2;
     T1 := BLANKS(5);
     T2 := BLANKS(5);
     T1 := INTEXT(5);
     T2 := INTEXT(2);
```

Si on a une carte de données : ABCDEFGH

On aura comme résultats T1 = ABCDE.
T2 = FG.

Un programme étant appelé à travailler avec un nombre variable (et non toujours connu) de données, il est utile de pouvoir vérifier si on a atteint la fin de notre fichier d'entrée.
Deux procédures LASTITEM et ENDFILE nous permettent de tester cet état de choses.

boolean procedure LASTITEM ;

Cette procédure prend la valeur vraie s'il n'y a plus d'autres caractères que des blancs dans le reste du fichier. Un appel à LASTITEM fait avancer "POS" jusqu'au prochain caractère non-blanc (et alors LASTITEM est faux) ou jusqu'au end-of-file (et alors LASTITEM est vrai).
Ex: on veut lire des entiers, les mettre dans un vecteur avant d'effectuer des calculs :

```
integer array T [1 : 1000];
integer I;
I := 0
for I := I+1 while      ¬   LASTITEM
              do  T [I] := ININT ;

CALCUL :      .
              .
              .
```

boolean procedure ENDFILE ;

Cette procédure prend la valeur vraie si on a essayé de lire une nouvelle carte, alors que toutes les cartes de données ont été lues.
Ex: on veut imprimer les cartes de données :

```
                    text T;
                    T := BLANKS(80);
                    while ¬ENDFILE do begin
                                      T := INTEXT(80);
                                      INIMAGE
                                      end;
FIN :               .
                    .
                    .
```

3.2.3. Procédures de sortie

Au début du programme, le tampon est initialisé avec des blancs et "POS" pointe le premier caractère du tampon.
Il existe cinq procédures correspondant aux procédures d'entrée :

OUTIMAGE : permet d'imprimer le contenu du tampon de sortie. Celle-ci transfère le contenu du tampon sur une ligne d'imprimante, restaure les blancs dans le tampon, remet "POS" à 1, de telle sorte qu'une nouvelle ligne pourra être reconstruite dans celui-ci de gauche à droite.

S'il n'y a pas d'appel à OUTIMAGE, la prochaine sortie se fera dans le tampon courant à partir de l'endroit où "POS" a été laissé.
Dans toutes les procédures de sortie, un OUTIMAGE est exécuté automatiquement par le système si le nombre de caractères requis pour l'édition du nombre ne peut être contenu dans le tampon courant et l'édition se fera dans le nouveau tampon vide.

OUTCHAR(C) : C est le caractère à éditer. Ce caractère sera écrit dans le tampon au caractère pointé par "POS". "POS" sera augmenté de 1.

OUTINT(I;W) : I est l'entier à éditer et W la longueur du champ (en caractères) dans lequel l'entier est écrit dans le tampon à partir du caractère pointé par "POS". Les nombres sont cadrés à droite.

OUTFIX(X,R,W) : sortie d'un flottant X où R est le nombre de caractères réservés à la partie décimale et où W est la longueur du champ en caractères dans lequel le nombre doit être décrit dans le tampon.

OUTTEXT(T) : T représente le texte. Le champ est défini par la longueur du texte. Il commence au caractère pointé par "POS".
Après l'édition, "POS" pointera le caractère suivant le texte.

Il serait utile au programmeur de pouvoir fixer la position de "POS" dans le tampon avant de lire ou écrire.
La procédure SETPOS(N) permet de pointer au $N^{ième}$ caractère du tampon.
Mais une procédure comme SETPOS(N) peut s'appliquer à n'importe quel fichier; il faut trouver un moyen d'indiquer duquel il s'agit.
En SIMULA, cela peut être résolu par la notation avec point, c'est-à-dire en préfixant la procédure par le nom du fichier voulu : SYSIN représente le fichier d'entrée et SYSQOUT le fichier de sortie. SYSIN et SYSOUT sont les noms donnés par le système aux fichiers standards.

On peut aussi vouloir connaître la position du pointeur "POS" dans chacun des fichiers. Un appel à la procédure "POS" permet de savoir à quel endroit du fichier on se trouve.

3.3. NOTION DE CLASSE

La puissance de SIMULA réside particulièrement dans l'utilisation du concept de CLASS.
En apparence, dans sa définition, une classe ressemble beaucoup à une procédure. On peut, en effet, se servir d'une classe comme d'une procédure mais le concept rejoint aussi ceux de structure (1) et de coroutine (2).

Les principaux avantages des classes sont les suivants :

(1) Gestion de mémoire :

Les classes permettent au programmeur de faire l'allocation de blocs de mémoire et de les libérer selon ses besoins.

(2) Parallélisme :

L'exécution d'une classe peut différer de celle d'une procédure et se faire d'une façon quasi-parallèle.

(3) Extension du langage :

Les classes peuvent servir à définir de nouveaux types de variables à partir des types déjà définis.

Toutes ces propriétés font des classes un outil qui facilite beaucoup le traitement des listes et la simulation.

3.3.1. Classe et Objet

Une déclaration de classe se fait de la même façon qu'une déclaration de procédure. On trouvera donc un nom de classe, des paramètres et leurs spécifications s'il y a lieu, suivis d'un bloc définissant des variables locales et une séquence d'instructions.

```
procedure BIDON(A);
integer A;
begin
        integer B;
        B := A
end;
```

```
class BIDON(A);
integer A;
begin
        integer B;
        B := A;
end;
```

L'appel d'une procédure implique la création d'un bloc temporaire pour contenir les variables locales de la procédure et ses paramètres. Appelons "bloc d'activation" ce bloc temporaire. L'accès aux variables d'un bloc d'activation (variables locales) est strictement limité aux instructions à

l'intérieur de la procédure correspondante. A la fin de l'exécution d'une procédure, l'accès à ces variables n'est plus possible et la mémoire occupée par le bloc d'activation peut être rendue au système.
Il est toutefois possible d'avoir pour une même procédure plusieurs blocs d'activation présents en même temps en mémoire : procédures récursives. A propos de celles-ci, il est utile de considérer que chaque appel crée une copie intégrale de la procédure et que le bloc d'activation contient non seulement des variables mais aussi une copie locale du programme de la procédure.

En réalité, les instructions pour une procédure sont codées de manière réentrante et sont placées à un seul endroit en mémoire. Pour donner à chaque activation sa "copie locale", il suffit d'ajouter aux variables du système du bloc d'activation un compteur ordinal où il faudra continuer.

Un appel de class donne lieu lui aussi à la création d'un bloc d'activation.
En SIMULA, ce bloc a un nom et s'appelle OBJET. Ces objets, à la différence des blocs d'activation des procédures, ont une certaine permanence et ne sont pas rendus au système après la fin de l'exécution des instructions de la classe. Il est aussi possible d'avoir accès aux variables locales d'un objet à partir d'instructions non-locales à la classe.
Il reste à voir comment créer ces objets, comment les distinguer les uns des autres, comment avoir accès à leurs variables locales et comment éventuellement, s'en débarrasser.

3.3.2. Création d'un objet

Un appel de la classe (par exemple BIDON) se fait par un énoncé de la forme : PT :- new BIDON(3) : et qui aura pour effet :

1) de créer un objet de la classe BIDON. Cet objet aura ses variables locales telles que décrites dans la déclaration de classe et son programme sera identique à celui de la classe, il y aura copie en mémoire de la classe BIDON. Donc, l'objet sera exécuté, ne servant que de modèle.
2) d'évaluer les paramètres effectifs et d'initialiser les variables locales de l'objet à des valeurs "nulles".

3) de donner le contrôle à l'objet pour l'exécution de ses énoncés.
4) new BIDON(3) prendra la valeur de l'adresse de l'objet pour l'affecter au pointeur PT.

Après l'exécution de l'objet, le contrôle retourne donc au programme appelant comme dans le cas d'une procédure. Cependant, le pointeur PT contient maintenant l'adresse de l'objet créé et il sera possible par la suite d'atteindre les variables locales du nouvel objet en passant par PT (qui sera de type pointeur).

3.3.3. Pointeurs

Un pointeur est un type de variable au même titre qu'un réel ou un entier. En SIMULA, les pointeurs contiennent l'adresse des objets créés durant l'exécution et servent à nommer ces objets et enfin les rendre accessibles sans équivoque.

3.3.3.1. Déclaration de pointeur

Chaque pointeur doit être déclaré et les déclarations sont placées au même endroit que les déclarations des variables numériques ou logiques.
Une déclaration se fait de la façon suivante :

ref (nom de classe) liste de noms de pointeurs ;

Ex: ref (BIDON) PT ;
ref (HOMME) PIERRE,JEAN,JACQUES ;

Un pointeur ne peut pointer que vers des objets d'une seule classe, celle indiquée dans la déclaration. Cette déclaration ne crée par les objets de la classe associée; elle ne fait que réserver l'espace pour les pointeurs.
Au début du programme, les pointeurs ne pointent donc sur aucun objet. Cette notion du pointeur vide est formalisée en ajoutant à SIMULA une constante none qui est l'adresse d'un objet vide de classe universelle vers lequel tous les pointeurs lcoaux sont initialisés avec cette valeur none.

3.3.3.2. Opération sur les pointeurs

Les opérations sont limitées à l'affectation et la comparaison d'égalité et d'inégalité. Pour affecter une valeur à un pointeur, on utilise un énoncé de la forme :

nom de pointeur :- valeur de pointeur ;

(la notation ":=" est réservée à l'affectation de valeurs autres que celles de type "adresse").

L'exemple suivant montre l'utilisation de classes et de pointeurs. Dans ce programme, il n'y a qu'une classe qui décrit la représentation d'un "homme". Les objets de type "homme" sont caractérisés par deux variables locales donnant leur âge et indiquant s'ils sont vieux. L'âge est passé comme paramètre et la vieillesse est déterminée par le programme local de la classe.

```
 1  begin
 2  ref (HOMME) PIERRE,JEAN,JACQUES ;
 3  class HOMME(AGE) ; integer AGE ;
 4  begin
 5              boolean VIEUX ;
 6              VIEUX := AGE > 20
 7  end ;
 8  JEAN :- new HOMME(25) ;
 9  JACQUES :- new HOMME(4) ;
10  PIERRE :- JEAN ;
11  JEAN :- none ;
12  end ;
```

Le programme crée deux objets distincts correspondant aux deux appels de classe new HOMME(25) et new HOMME(4).
Après la création, le contrôle est donné au nouvel objet qui calcule la valeur de son booléen local "VIEUX" avant de retourner le contrôle au programme principal. Après l'énoncé 9, les adresses de ces deux objets se trouvent dans "JEAN" et "JACQUES". Ces pointeurs, en quelque sorte, servent de noms aux objets et permettent de les distinguer.

Un pointeur ne peut désigner qu'un seul objet à la fois, mais plusieurs pointeurs peuvent désigner le même.

Après l'énoncé 10, PIERRE et JEAN se réfèrent au même objet HOMME(25). Ainsi, on peut vouloir examiner si deux pointeurs sont affectés au même objet (s'ils sont égaux). Les opérateurs logiques utilisés pour comparer deux pointeurs sont == pour l'égalité et =/= pour l'inégalité.

Ex: if PIERRE == JEAN then begin ... end ;

L'énoncé 11 coupe la relation entre NOM et OBJET en affectant la valeur none à JEAN. Donc, même si l'homme de 25 ans a été créé sous le nom de JEAN, il faudra désormais l'atteindre avec le nom de PIERRE. Si on affecte none à tous les pointeurs les adresses des deux objets créés sont perdues et ces objets ne seront plus accessibles par le programme.
Au point de vue programme, ces objets n'existent plus. La garbage collection de SIMULA récupérera, en temps et lieu, la mémoire occupée par ces objets, pour la rendre au système.

3.3.3.3. Auto-référence

Il est souvent utile à un objet de pouvoir récupérer sa propre adresse sans avoir recours à des pointeurs globaux. Cette auto-référence est donnée par :

this X où X est un nom de classe approprié.

La qualification d'une auto-référence telle que this X est donnée par le nom de classe qui suit this.
Ceci permet à un objet d'avoir accès à sa propre adresse.

3.3.4. Accès aux objets

Pour atteindre les variables internes d'un objet, SIMULA se sert d'une *notation avec point*, combinant le nom de la variable avec le pointeur vers l'objet voulu.
Supposons qu'on veuille savoir l'âge de JACQUES (cfr exemple précédent).
On utilisera un énoncé de la forme : if JACQUES.AGE > 50 then begin

...

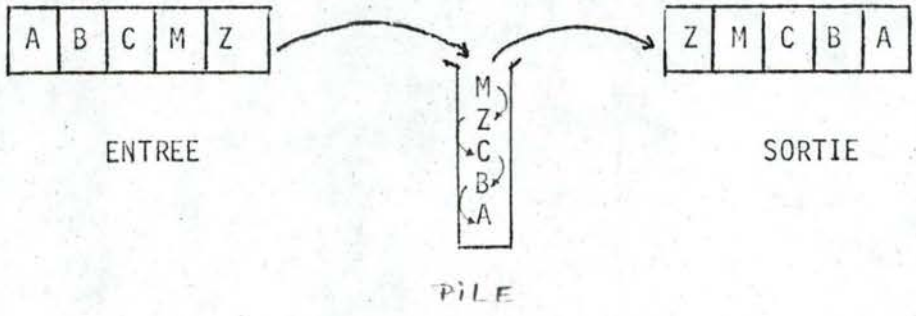
end ;

On peut utiliser la notation avec point de façon itérative.
En SIMULA, ces identificateurs non-locaux peuvent prendre la place d'identificateurs simples dans n'importe quelle expression. S i on veut vieillir JACQUES, on aura la séquence suivante :

```
JACQUES.AGE := JACQUES.AGE  +  30 ;
JACQUES.VIEUX :=  JACQUES.AGE > 20 ;
```

3.3.5. Utilisation de classe

Pour démontrer l'utilisation des classes, nous allons aborder un exemple et le programme de deux manières différentes.
L'exemple est simple : on lit une chaîne de caractères de longueur non définie et on veut l'écrire en sens inverse.
Pour chaque carrière lu on va créer un objet qui contiendra le caractère.
Le processus est décrit dans le dessin suivant :



Classe sans paramètres ni instructions

```
begin
      ⌈class ELEMENT ;
      |begin
      |       character C ;
      |      ref(ELEMENT) PT
      |end  de la classe ;
character CAR ;
ref (ELEMENT) PILE, P ;
while     LASTITEM  do  begin
                              CAR := INCHAR ;
                              P := new ELEMENT ;
                              P.C := CAR ;
                              P.PT ;- PILE ;
                              PILE :- P
                        end ;
while  PILE  =/=  none  do  begin
                                 OUTCHAR(PILE.C) ;
                                 PILE :- PILE.PT
                            end ;
end ;
```

Classe avec paramètres mais sans instructions

```
begin
      ⌈class ELEMENT(C,PT) ;
      |character C ;
      |ref(ELEMENT) PT ;   ;
ref(ELEMENT) PILE ;
while   LASTITEM  do  PILE :- new ELEMENT(INCHAR,PILE) ;
while   PILE  = / = none  do  begin
                                  OUTCHAR(PILE.C) ;
                                  PILE :- PILE.PT
                              end
end ;
```

3.3.6. Procédures de contrôle d'exécution (quasi-parallèle)

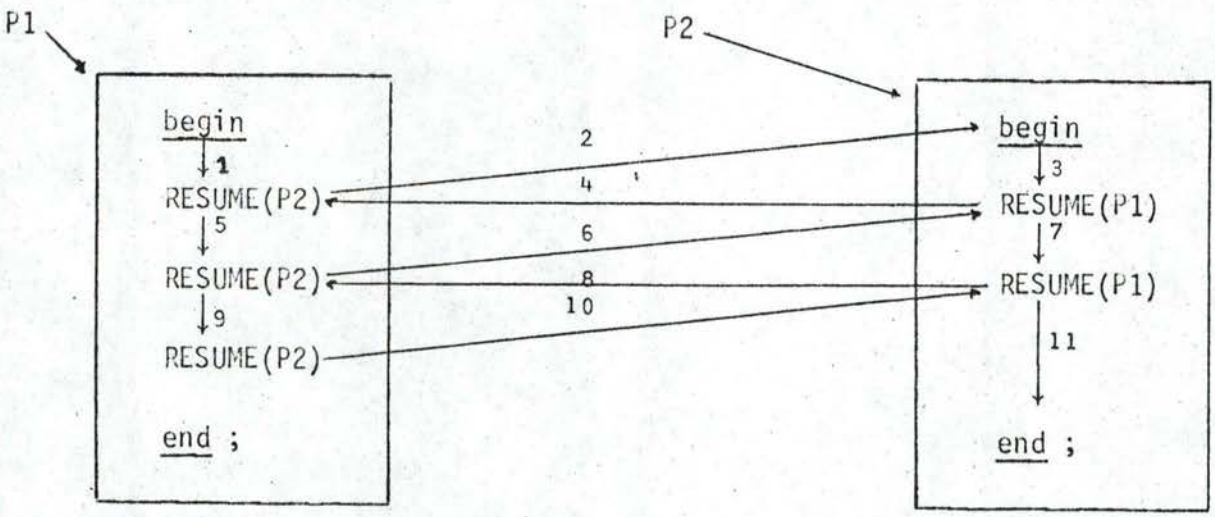
Jusqu'ici, tous les objets considérés ont été exécutés en entier à leur création, c'est-à-dire jusqu'à leur end final pour devenir des objets terminés. Cette utilisation de classe ne met en évidence qu'un aspect des avantages des classes sur les procédures. En effet, les objets correspondant à une classe peuvent être exécutés par phases, à l'aide des procédures RESUME(PT) et DETACH.

RESUME(PT)

Cette procédure accomplit trois actions :

- elle interrompt le cours normal de l'exécution;
- elle garde l'adresse de la prochaine instruction à exécuter dans l'objet où le RESUME se produit;
- elle passe le contrôle à l'objet pointé par PT.

Cette procédure permet l'interaction entre différents objets pour une exécution quasi-parallèle. Deux objets pointés par P1 et P2 seront exécutés de la façon suivante :



Voyons maintenant comment on peut utiliser la procédure RESUME(PT). On désire programmer un jeu auquel participent plusieurs joueurs (un jeu de dés par exemple). Chaqun joue à son tour selon les mêmes règles. On peut donc représenter chaque joueur par un objet d'une classe JOUEUR, s'exécutant de manière quasi-parallèle. On peut définir cette classe de la façon suivante :

```
                class  JOUEUR ;
                begin
        LOOP :  déterminer la stratégie :
                jouer ;
                RESUME(PROCHAIN) ;
                goto  LOOP
                end ;
```

Supposons qu'il y ait quatre participants qui jouent à tour de rôle :

```
begin
integer I ;
ref(JOUEUR) array A [1:4 ];
class JOUEUR(N) ; integer N ;
begin
      ref(JOUEUR) NEXT ;
      NEXT :-  if N=4  then A [1]  else A [N+1 ];
LOOP :  .
        .
        .
      RESUME(NEXT) ;
        .
        .
      goto LOOP ;
      end  ;
for  I := 1  step 1  until 4  do
A [ I] :- new JOUEUR(I) ;
RESUME(A[ 1] )
end ;
```

L'utilisation de RESUME entraîne cependant un problème à la création de l'objet. En effet, à sa création, A[1] sera exécuté et tentera donc de donner le contrôle à l'objet pointé par A [2] (par l'énoncé RESUME(NEXT)) mais, à ce moment ce joueur n'a pas été encore créé.

Pour résoudre ce problème, il faudra arrêter momentanément l'exécution de A [1] pour pouvoir créer A [2]. Ceci peut se faire à l'aide de la procédure DETACH.

DETACH

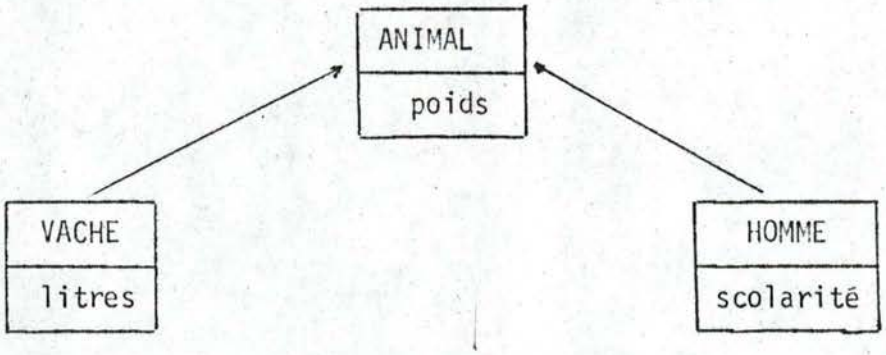
Cette procédure accomplit quatre actions :

- elle interrompt l'exécution de l'objet présentement en cours;
- elle garde l'adresse de l'énoncé suivant le DETACH dans cet objet;
- elle redonne le contrôle à l'objet qui a créé l'objet où se produit le DETACH, à l'énoncé suivant la dernière interruption de l'exécution de celui-ci;
- elle marque l'objet comme étant "détaché".

Dans l'exemple précédent, il suffit donc de placer l'énoncé DETACH comme première *instruction* de la classe JOUEUR. Les objets détachés pourront être exécutés par la suite par un appel à la procédure RESUME(PT).

3.3.7. Classe préfixée - INNER

En SIMULA, on indique qu'une classe est une sous-classe d'une autre en mettant le nom de la classe commune en préfixe :
Par exemple, prenons trois classes ; ANIMAL, VACHE et HOMME.




```
        class ANIMAL ;
        begin
        end ;     real POIDS

ANIMAL  class VACHE ;
        begin
        end ;     real LITRES

ANIMAL  class HOMME ;
        begin
        end ;     integer SCOLARITE
```

En préfixant une définition de classe avec le nom d'une autre classe, on obtient la *concaténation* des deux définitions. Les objets de la classe ainsi préfixés obtiennent tous les attributs de la classe servant de préfixe. Un objet qui est membre d'une sous-classe est automatiquement membre de la sur-classe servant de préfixe mais l'inverse n'est pas vrai. Donc, un objet de type VACHE est aussi membre de la classe ANIMAL; mais tout ANIMAL n'est pas nécessairement une VACHE, certains seront des hommes et d'autres ne seront ni HOMME ni VACHE.

Voyons de manière plus générale ce qui se passe quand une classe est préfixée par une autre.

Soit une classe CLUN préfixée par une classe CLZERO, chacune ayant ses paramètres PAR, ses déclarations DCL et ses instructions INST comme suit :

```
        class  CLZERO(PARZERO) ; ... spécifications de PARZERO ... ;
        begin
              - DCLZERO -
              - INSTZERO -
        end ;

CLZERO  class  CLUN(PARUN) ;  ... spécifications de PARUN ... ;
        begin
              - DCLUN -
              - INSTUN -
        end ;
```

Du fait qu'elle est préfixée par CLZERO, la classe CLUN est équivalente à

```
        class  CLUN(PARZERO,PARUN) ;
                 ...  spécifications de PARZERO et PARUN ... ;
        begin
              - DCLZERO -
              - INSTZERO -
          begin
                    - DCLUN -
                    - INSTUN -
          end ;
        end ;
```

Les objets de la classe CLUN ont les variables locales définies dans DCLZERO et DCLUN et exécutent les instructions INSTUN. Le bloc interne indique qu'il y a une limite sur la portée des identificateurs et que les instructions de la sur-classe n'ont pas accès aux variables de la sous-classe. A la création d'un objet de CLUN, il faut fournir PARZERO et PARUN.

Un objet d'une classe B préfixée par une classe A a un programme local composé des instructions de A suivies des instructions de B. En général, les instructions de A servent à l'initialisation des variables. Parfois il serait utile de définir aussi dans A des actions à effectuer après l'exécution des instructions de B. Ceci est possible à l'aide du mot-clé inner.

inner indique où placer les instructions d'une sous-classe par rapport aux instructions d'une sur-classe. inner évidemment ne peut apparaître qu'une seule fois dans le corps d'une classe.

Exemple :

```
class A ;
begin
        - INSTA -
        inner ;
        - INSTC -
end ;
```

Si cette classe sert de préfixe à la classe B :

```
A class B ;
  begin
  end ;    - INSTB -
```

Dans l'objet concaténé A+B, inner sera remplacé par les instructions de B :

```
class B ;
begin
        - INSTA -
        begin
                - INSTB -
        end ;
        - INSTC -
end ;
```

L'absence de inner équivaut à un inner placé comme dernière instruction d'une définition de classe.

3.3.8. Enoncé INSPECT

L'énoncé inspect qui sera utile pour le traitement de liste permet à la fois de trouver la qualification d'un objet quelconque et aussi d'avoir accès aux variables locales sans utiliser la notation avec point.
On verra qu'une liste pourra contenir des membres correspondant à des classes différentes. Il est parfois nécessaire de savoir à quelle classe chacun des membres appartient.
Plusieurs formes sont possibles :

```
inspect pointeur do
                 :
                 : ;

inspect pointeur when nom-de-classe do begin ... end
                when nom-de-classe do begin ... end
                .
                .
                otherwise begin ... end ;
```

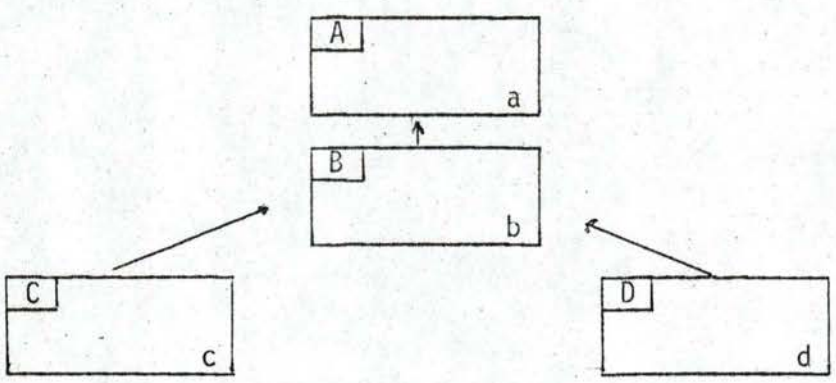
Nous avons dit précédemment qu'un pointeur ne peut désigner que des objets de la classe spécifiée dans la déclaration. La possibilité d'avoir des sous-classes modifie un peu les choses et on permet qu'un pointeur désigne aussi des objets des sous-classes de la classe indiquée.

Explicitons ceci par un exemple :

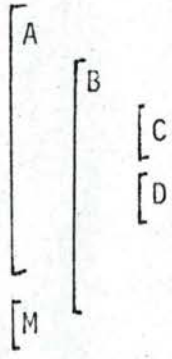
```
    class A ; ... integer a ; ... ;
A   class B ; ... integer b ; ... ;
B   class C ; ... integer c ; ... ;
B   class D ; ... integer d ; ... ;
    class M ; ... integer m ; ... ;
```

avec
```
ref(A) PA ;
ref(B) PB ;
ref(C) PC ;
ref(D) PD ;
ref(M) PM ;
```

La figure qui suit montre deux manières de représenter l'ensemble des définitions de l'exemple donné :



Structure d'arbre



Structure de bloc

La structure d'arbre indique bien la hiérarchie des définitions. La structure de bloc nous rappelle les limites sur les champs d'instruction : le champ de D, par exemple, comprend les variables locales de A, B et D mais non celles de C ni celles de M.

Dans cet exemple, il y a cinq types d'objets possibles : A B C D et M. Trois de ces types sont des objets composés appartenant à des classes préfixées. Ces objets contiennent tous les attributs des classes préfixées; ils sont aussi membres de plusieurs classes à la fois. Le tableau indique la composition et les classes auxquelles appartiennent ces cinq types d'objets :

Objet	Méthode de création	Objet créé (variables locales)	Qualification	Appartenance à la classe A	B	C	D	M
1	PA :- new A	A (a)	A	v				
2	PA :- new B	A+B (a+b)	B	v	v			
3	PA :- new C	A+B+C (a+b+c)	C	v	v	v		
4	PA :- new D	A+B+D (a+b+d)	D	v	v		v	
5	PA :- new M	M (m)	M					v

v ≡ MEMBRE

A la création d'un objet avec new X, l'objet est composé de toutes les classes rencontrées sur la branche qui relie X à la racine de l'arbre de définition.

La qualification d'un objet est le nom de la sous-classe la plus inférieure de l'objet.
Un objet qui a la qualification X est membre de toutes les classes rencontrées sur la branche reliant X à la racine.

Prenons l'énoncé suivant qui se sert de la notation avec point :

X := pointeur.variable ;

Pour s'assurer que l'énoncé est correct, il faut vérifier que la qualification de l'objet désigné par le pointeur est telle que "variable" en soit un attribut. Il est possible à la compilation de vérifier si "variable" est bien dans le champ des objets désignés par "pointeur". Le seul test qui doit être fait à l'exécution vérifie que le pointeur ne désigne pas l'objet vide none.

En bref, l'expression "pointeur.variable" est acceptée lors de la compilation si "variable" est dans le champ de la classe qualifiant "pointeur".

Voici quelques exemples d'expressions :

acceptées	refusées
PA.a	PA.d
PD.d	PA.m
PC.b	PD.c

3.4. TRAITEMENT DE LISTE ET SIMSET

On peut définir une liste comme étant un ensemble fini et ordonné d'éléments. La liste est délimitée par des parenthèses et les éléments constituants sont représentés par des identificateurs séparés par des virgules. Par exemple, on retrouve :

Pour la liste

EMPTY qui fournit un booléen indiquant si la liste est vide.
FIRST, LAST qui désignent respectivement le premier et le dernier éléments d'une liste.
CLEAR qui vide la liste.

Pour les éléments

SUC, PRED qui désignent les voisins d'un élément.
OUT qui extrait un élément d'une liste.
INTO(LISTE) qui insère un élément à la fin de LISTE.

Soit LISTE qui désigne la liste (A,B,C,D) et ELEMENT qui désigne l'élément A. En prenant la notation avec point pour indiquer les oéprations :

LISTE.FIRST	donne A
LISTE.LAST	donne D
LISTE.EMPTY	donne false
ELEMENT.SUC	donne B
ELEMENT.PRED	donne none
ELEMENT.OUT	donne LISTE = (B,C,D), ensuite, si nous
faisons ELEMENT.INTO(LISTE)	cela nous donne LISTE = (B,C,D,A).

Le concept de liste est implanté en chaînant des objets à l'aide de pointeurs.

On va d'abord voir quelques définitions possibles en SIMULA pour le concept de LISTE ainsi que pour les opérations sur les listes.
Une implantation est suggérée et les définitions se trouvent sous forme précompilée dans une classe du système.
L'usager peut s'en servir en préfixant son programme avec SIMSET le nom de cette classe.

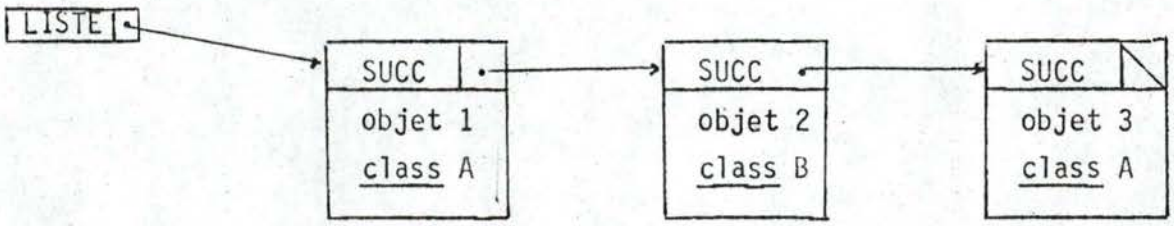
3.4.1. Structures de listes

Dans cette section, seront examinés les objets utiles et nécessaires pour décrire les listes ainsi que les procédures classiques de traitement de listes.

Au départ, une structure simple sera donnée qui sera modifiée peu à peu pour aboutir à une structure très proche de celle de SIMSET.

Liste à un sens

La liste la plus simple est la liste à un sens avec un pointeur de début de liste. On peut la représenter ainsi :



Chaque objet doit appartenir à la classe A ou à la classe B. De plus, chaque objet doit contenir une variable additionnelle SUCC qui lui permet de repérer son successeur. Cette structure de liste a certains inconvénients reliés à sa simplicité. Par exemple, étant donné un pointeur vers un élément, il est impossible soit de trouver son prédécesseur ou de l'extraire et de raccorder la liste, à moins d'avoir en plus un pointeur vers le début de la liste. Néanmoins, cette structure sert de point de départ pour notre discussion.

Les objets de la liste pourraient être définis de la façon suivante :

```
class  A ;
begin
       ref(B) SUCC ;
        :
end ;

class  B ;
begin
       ref(A) SUCC ;
        :
end ;
ref(A)  LISTE ;
```

La liste doit toujours commencer par un objet de la classe A puisqu'on a ref(A) LISTE. De plus, seul un B peut suivre un A et seul un A peut suivre un B. Il est aussi lourd d'avoir à définir le pointeur SUCC pour chaque définition de classe.

La classe LINK

Pour résoudre ce problème, il suffit de définir une classe spéciale que nous appellerons LINK qui devra préfixer tout objet susceptible d'être dans une liste. Voici une définition possible de LINK :

```
class  LINK ;
begin
        ref(LINK)  SUCC ;
end ;
```

Les autres déclarations deviendraient :

```
                ref(LINK)  LISTE ;
LINK    class   A ;   begin  ...  end ;
LINK    class   B ;   begin  ...  end ;
```

Un objet dans une liste n'a pas besoin de savoir, a priori, la classe exacte qui le suit; il lui suffit de savoir que l'objet est un LINK. On voit l'utilité de la déclaration hiérarchique des classes.

La classe LINK est aussi un bon endroit pour mettre les procédures associées aux *éléments* d'une liste :

SUC : fournit un pointeur vers le successeur d'un élément si celui-ci est dans une liste et none autrement.

MEMBRE : un prédicat qui rapporte un booléen indiquant si l'objet est dans une liste.

FOLLOW(X) : qui insère un élément non-membre d'une liste à la suite de X si X est membre d'une liste.

Pour les implanter, il est nécessaire d'adopter une convention pour le contenu de SUCC. Si l'objet n'est pas membre d'une liste, SUCC pointera sur l'objet lui-même. Autrement, SUCC désigne le successeur ou none si l'objet est le dernier de la liste.

Voici la définition de LINK avec procédures :

```
class  LINK ;
begin
ref(LINK)  SUCC ;

boolean procedure  MEMBER ;
   MEMBER :=  SUCC =/=  this  LINK ;

ref(LINK) procedure  SUCC ;
   SUCC :- if   MEMBER  then  SUCC  else  none ;

procedure  FOLLOW(X) ;  ref(LINK) X ;
   if  X ==  none  then  ERREUR  else
    if    MEMBER   X.MEMBER  then  SUCC :- X.SUCC :- this LINK

SUCC :- this LINK
end ;
```

La classe HEAD

Cette classe contiendra les procédures de *liste*, tout comme LINK contient les procédures pour les éléments. HEAD contiendra en plus un pointeur SUCC, pointant vers le premier objet de la liste. Voici une définition possible de HEAD avec les procédures : EMPTY, FIRST et LAST.

```
class  HEAD ;
begin
ref(LINK)  SUCC ;

boolean procedure  EMPTY ;
   EMPTY  :=  SUCC == none ;

ref(LINK) procedure  FIRST ;
   FIRST  :- SUCC ;

ref(LINK) procedure  LAST ;
   if  SUCC == none   then  LAST :- none
                      else  begin
                ref(LINK) PT ;
                PT :- SUCC ;
                while PT.SUCC =/= none do  PT :- PT.SUCC ;
                LAST :- PT
                                  end
end ;
```

la seule procédure un peu compliquée est celle pour LAST à cause de la liste qui permet se lement le parcours dans un sens.
Pour créer une liste, il suffit de créer un nouveau HEAD avec un énoncé du type

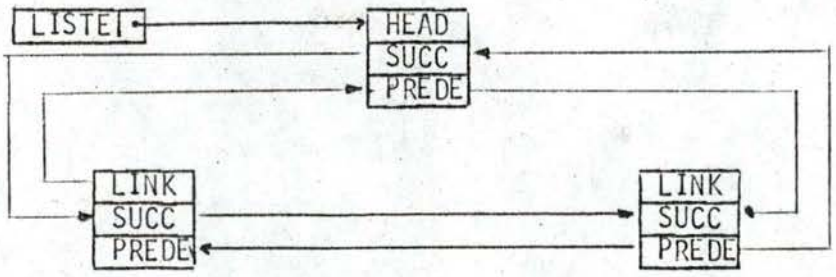
```
LISTE   :- new HEAD ;
```

Avant de continuer, il serait utile de définir la procédure INTO(LISTE) qui doit être placée dans LINK et qui met un elément à la fin d'une lsite. Il n'était pas facile de décrire cette procédure plus tôt car HEAD n'était pas encore défini.

```
procedure  INTO(LISTE) ; ref(HEAD)  LISTE ;
if   MEMBER   LISTE =/= none  then  begin
   if  LISTE.EMPTY  then  begin
       LISTE.SUCC :-  this  LINK  ;
       SUCC  :- none
                              end
                    else  FOLLOW(LISTE.LAST)
                                        end ;
```

Liste à deux sens

Certaines des procédures de liste classiques décrites au début du chapitre ne sont pas faciles à implanter avec une liste à un sens. La procédure LAST est très inefficace et il est impossible de trouver le prédécesseur d'un élément ou de sortir un élément d'une liste sans avoir aussi accès à objet HEAD de la liste.
Pour remédier à cette situation, il faut ajouter un pointeur PREDE aux définitions de HEAD et LINK.
Dans le cas de LINK, PREDE désigne le prédécesseur et, pour HEAD, PREDE désigne le dernier élément. Une liste a maintenant la structure suivante :



La liste forme donc un cercle. HEAD se trouve à avoir les mêmes pointeurs que les éléments et c'est pourquoi on leur a donné des noms identiques. Avec cette structure, il y a un problème de qualification, car les pointeurs SUCC et PREDE d'un élément désignent parfois un LINK et parfois un HEAD. Ce problème est facilement résolu en créant une nouvelle classe LINKAGE comprenant SUCC ET PREDE ainsi que les procédures SUC et PRED.
Cette classe est celle utilisée par SIMSET, l'ensemble des procédures et déclarations du système pour le traitement de liste.

Voyons maintenant d'une façon plus formelle comment ces principes sont réalisés par SIMULA et comment les utiliser.

3.4.2. Classe SIMSET

SIMSET, classe du système, contient toutes les facilités pour la manipulation de listes, à partir d'une structure circulaire telle que définie dans la section précédente. En effet, SIMSET met à notre disposition une structure à deux sens à l'aide des références SUCC et PREDE et chaque liste a une "tête de file" ou HEAD et un ensemble de membres. SIMSET contient donc trois classes LINKAGE, HEAD et LINK.
LINKAGE préfixe les deux autres car elle comprend des attributs communs à la classe HEAD et à la classe LINK.

3.4.2.1. Classe HEAD

Cette classe définit les attributs nécessaires à la tête de file qui ne sont pas définis dans la classe LINKAGE. Un ensemble vide ou liste vide ne

contient qu'un objet appartenant à cette classe.
Cinq procédures font partie de la classe HEAD :

ref(LINK) procedure FIRST : cette procédure est un pointeur à un objet de la classe LINK. Elle donne une référence au premier objet de l'ensemble des membres. Si la liste ne contient aucun membre, cette procédure prend la valeur none.

ref(LINK) procedure LAST : cette procédure fournit un pointeur au dernier objet de l'ensemble des membres. S'il n'y a aucun objet dans cet ensemble, elle prend la valeur none.

integer procedure CARDINAL : cette procédure permet de savoir combien il y a d'objets dans l'ensemble des membres.

boolean procedure EMPTY : cette procédure prend la valeur vrai si CARDINAL = 0, autrement EMPTY est faux.

procedure CLEAR : cette procédure vide l'ensemble des membres et la tête de file a son successeur et son prédécesseur pointant sur elle-même.

Ainsi la classe HEAD contient tout ce qui est nécessaire pour bien définir une liste. Lorsqu'on voudra créer une file, on définira un pointeur à la classe HEAD et ce pointeur sera le nom de la file.
Toutes les procédures vues plus haut seront accessibles à l'aide de ce pointeur avec la notation avec point.

3.4.2.2. Classe LINK

Cette classe contient toutes les procédures nécessaires à l'insertion et à l'extraction d'objets de l'ensemble des membres. Un objet pour faire partie de l'ensemble des membres doit être préfixé par LINK de façon à avoir accès aux quatre procédures définies dans celle-ci.
L'utilisation des procédures définies dans LINK se fait ordinairement à l'aide de la notation avec point.

procédure OUT : cette procédure prend l'objet dans l'ensemble où il se trouve et l'en extrait (un objet ne peut être que dans un seul ensemble à la fois).

procedure INTO(S) : ref(HEAD) S ; cette procédure prend l'objet (et l'enlève de l'ensemble où il se trouve, si nécessaire) et le place en queue de file dans l'ensemble des membres de la file pointé par S.

procedure FOLLOW(X) : ref(LINKAGE) X
procedure PRECEDE(X) : ref(LINKAGE) X

ces procédures enlèvent l'objet où il se trouve (si nécessaire), puis cherchent si X est dans un ensemble quelconque; si oui, l'objet devient successeur ou prédécesseur de X, selon le cas. Si X n'est dans aucun ensemble, ces deux procédures ont le même effet qu'un appel à OUT.

3.4.2.3. Classe LINKAGE

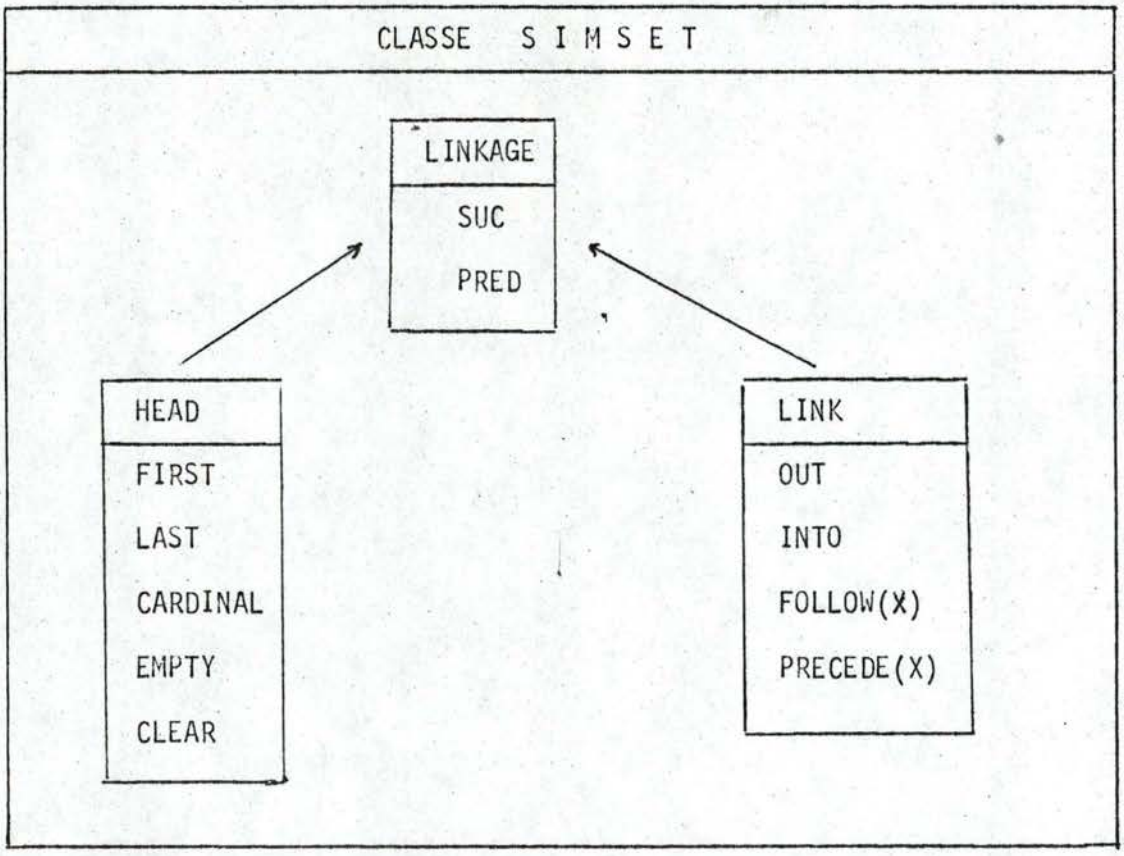
Cette classe permet d'atteindre le successeur et le prédécesseur pour tout élément d'une liste à l'aide de deux procédures.

ref(LINK) procedure SUC : cette procédure fournit un pointeur au successeur de l'objet considéré. Si ce successeur est la tête de file, SUC prend la valeur none.

ref(LINK) procedure PRED : cette procédure fournit un pointeur au prédécesseur de l'objet considéré. Si ce prédécesseur est la tête de file, PRED prend la valeur none.

Un tableau récapitulatif de la classe SIMSET nous donne les différentes procédures disponibles à l'utilisateur lors de la manipulation de listes. La classe SIMSET comprend trois classes : LINKAGE, HEAD et LINK. LINKAGE préfixe les deux autres classes.

(voir page suivante)




```
                class  SIMSET ;
                begin
                ...  classe linkage  ...
LINKAGE         class  HEAD ;        ...   ;
LINKAGE         class  LINK ;        ...   ;
                end de la classe  SIMSET ;
```

3.5. SIMULATION

3.5.1. Utilisation de la classe SIMULATION

SIMULA, pour venir en aide au programmeur, groupe dans une classe SIMULATION préfixée par SIMSET tous les outils nécessaires :

1) Une file (SQS) contenant des notices d'événements ordonnés chronologiquement.
2) Une classe PROCESS qui devra servir de préfixe à toute classe décrivant des objets pouvant changer l'état du système.
3) Une exécution chronologique de chaque événement contrôlée automatiquement.
4) Des énoncés d'ordonnancement spéciaux, ainsi que de nouveaux énoncés liés à l'exécution quasi-parallèle.
5) D'autres procédures et variables utiles (ex. TIME).

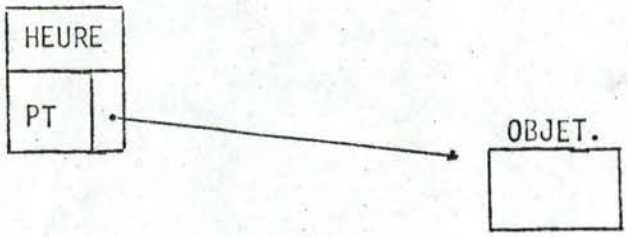
Si on préfixe un programme par SIMULATION, on aura accès à ces nouveaux éléments du langage en plus de ceux de SIMSET.

3.5.1.1. SQS (SEQUENCING SET)

La classe SIMULATION fournit tout d'abord une file circulaire de même type que celles décrites dans SIMSET, appelée SQS.
Cette file est automatiquement créée par le système au début d'un programme préfixé par SIMULATION. Les éléments contenus dans cette file (nommés notices d'événements) sont ordonnés chronologiquement par le système.
Le programmeur n'a pas un accès direct à cette file et à ses éléments, mais certains énoncés ou procédures lui donneront un contrôle sur ceux-ci.

3.5.1.2. Notices d'événement

Une notice d'événement comprend une variable indiquant l'heure à laquelle se produit un changement d'état du système (événement) et un pointeur provoquant le changement d'état :



Dès qu'un changement est cédulé, une notice d'événement est insérée dans SQS, à l'heure où il doit se produire.

3.5.1.3. PROCESS CLASS

Le pointeur d'une notice d'événement, comme tout autre pointeur, ne peut référer qu'aux objets d'une seule classe.
C'est pourquoi les pointeurs des notices d'événement réfèrent tous à une même classe appelée PROCESS dont la définition se trouve dans la classe SIMULATION.

Pour qu'un objet ait une notice d'événement dans SQS, la classe lui correspondant devra être préfixée par PROCESS. Ainsi, on peut appeler PROCESS tout objet provoquant un changement dans l'état du système simulé.

Mais comme on peut vouloir mettre un PROCESS dans une file d'attente, aussi la classe PROCESS est préfixée par LINK (on y a accès puisque SIMULATION est préfixée par SIMSET) permettant d'avoir accès aux procédures de traitement de liste (INTO, OUT, ...).

Lorsqu'un programme est préfixé par SIMULATION, le centre de contrôle de l'exécution se trouve dans SQS. En effet, à tout moment, l'objet qui est exécuté est celui qui est pointé en tête de file dans SQS; lorsque son exécution est terminée ou suspendue, sa notice d'événement est enlevée de SQS et le contrôle est donné au nouveau PROCESS dont la notice d'événement est maintenant en tête de file.

L'heure du système est celle qui correspond à l'heure comprise dans la notice d'événement en tête de file dans SQS.

Comme maintenant le contrôle de l'exécution se fait à partir de SQS, le programme principal est considéré comme un PROCESS.
Ainsi, au début de l'exécution, il n'y a qu'une notice d'événement dans SQS correspondant au programme principal et l'heure du système est zéro.

3.5.2. Enoncés particuliers à la classe SIMULATION

Comme on l'a déjà fait remarquer, le programmeur ne peut gérer lui-même directement la file SQS, ni créer les notices d'événement.
C'est pourquoi SIMULA met à sa disposition des énoncés et des procédures lui permettant de définir son algorithme de simulation pour le traitement automatique.

Enoncé ACTIVATE

Cette procédure permet d'insérer une nouvelle notice d'événement dans SQS. Une NE (notice d'événement) comprend un temps, un pointeur à un PROCESS et évidemment les pointeurs de la classe LINK indiquant les NE précédente et suivante.

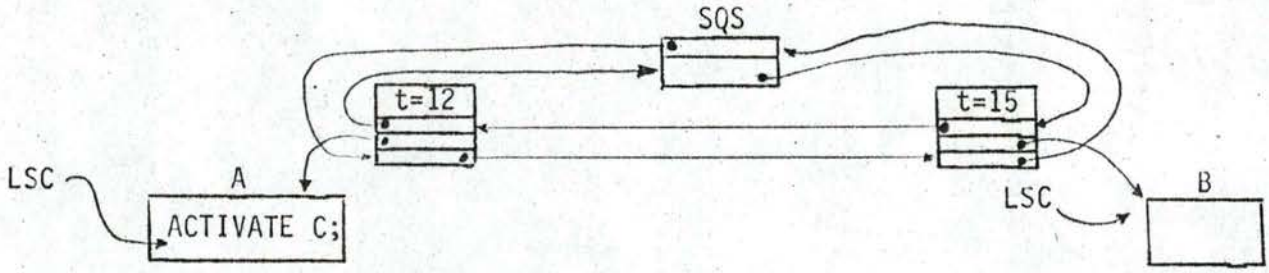
ACTIVATE X "position dans SQS" ; X peut être :
soit un pointeur à un PROCESS,
soit un new suivi d'un nom de PROCESS CLASS.

La position dans SQS est définie soit par un temps, soit relativement à une autre NE, soit les deux. ACTIVATE crée une NE référant à l'objet pointé par X et la place dans SQS à l'endroit voulu. Lorsque cette nouvelle NE arrivera en tête de SQS, le contrôle de l'exécution viendra et sera donné au PROCESS correspondant à l'endroit où son exécution avait été suspendue (un pointeur LSC indiquera cet endroit).

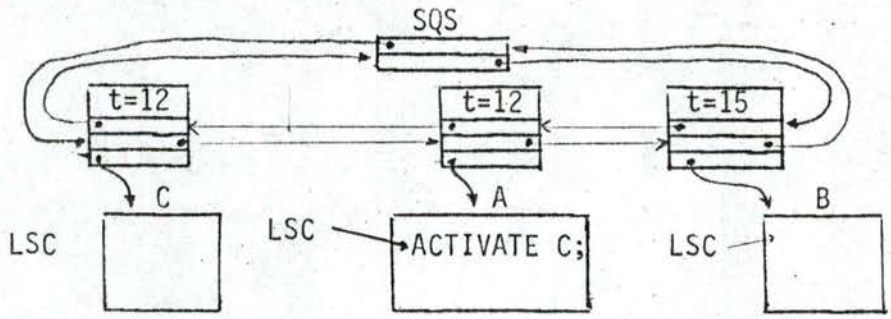
FORMES PARTICULIERES

ACTIVATE X la NE correspondant à X est placée *en avant* de la NE pointant l'objet courant, au même temps. Le contrôle de l'exécution est alors donné à l'objet X, et le LSC de l'objet qui était courant est placé à l'énoncé suivant ACTIVATE.

Exemple :

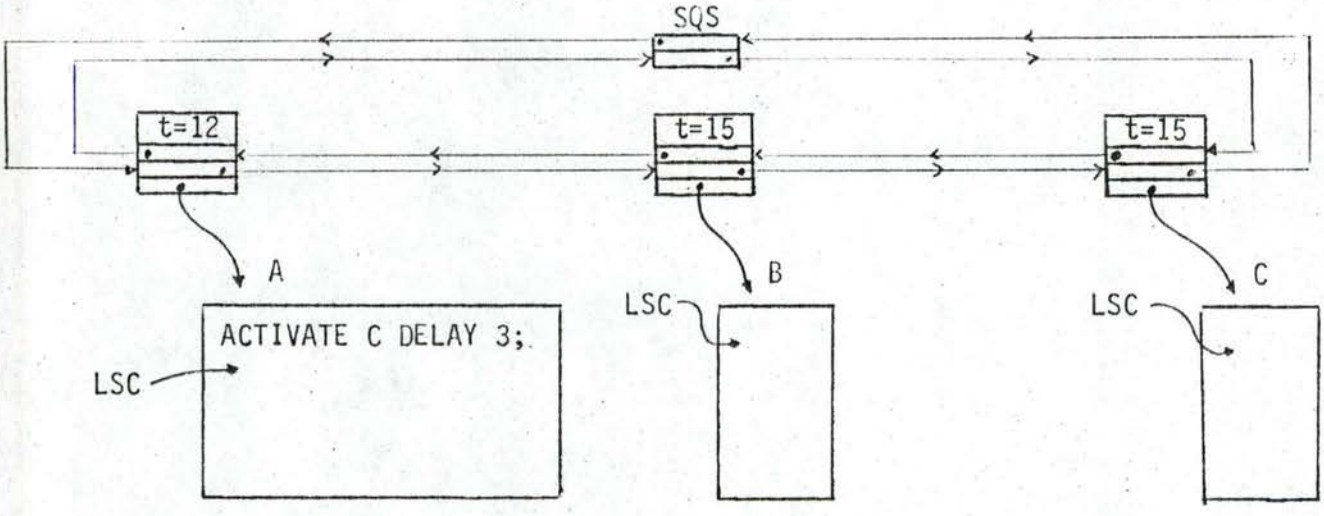


Après l'énoncé ACTIVATE dans A, on obtient :



ACTIVATE X DELAY T : permet de placer la NE correspondant à X au temps TIME+T à la suite de toutes les autres NE qui sont déjà cédulées au même temps.

Exemple : si on ajoute dans l'exemple précédent dans la classe A : ACTIVATE C DELAY 3, on aurait eu :



PRIOR : si on ajoute la particule PRIOR après T, la NE sera placée à TIME+T *devant* celles qui sont au même temps.

ACTIVATE X AT T : permet de placer la NE correspondant à X au temps T. Cet énoncé est équivalent à DELAY(T-TIME) avec les mêmes règles de priorité et on peut aussi utiliser la particule PRIOR (si le temps "schedulé" est plus petit que TIME, la NE sera placée au temps TIME sans priorité).

ACTIVATE X BEFORE Y : permet de placer la NE correspondant à X avant la NE se rapportant à Y dans SQS et du même temps que cette dernière. Y doit être une référence à un objet d'une PROCESS CLASS ayant une NE dans SQS.

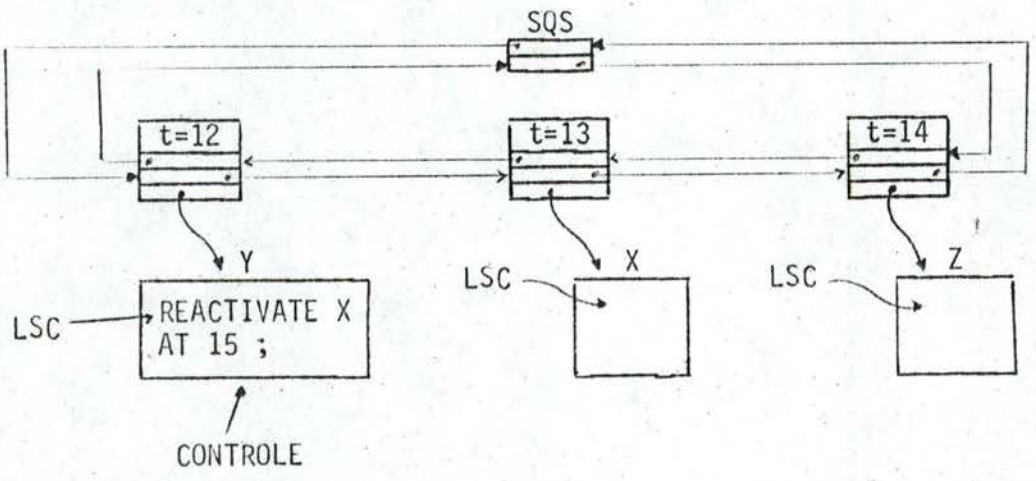
ACTIVATE X AFTER Y : même effet que BEFORE sauf que la NE correspondant à X se place immédiatement après celle de Y.

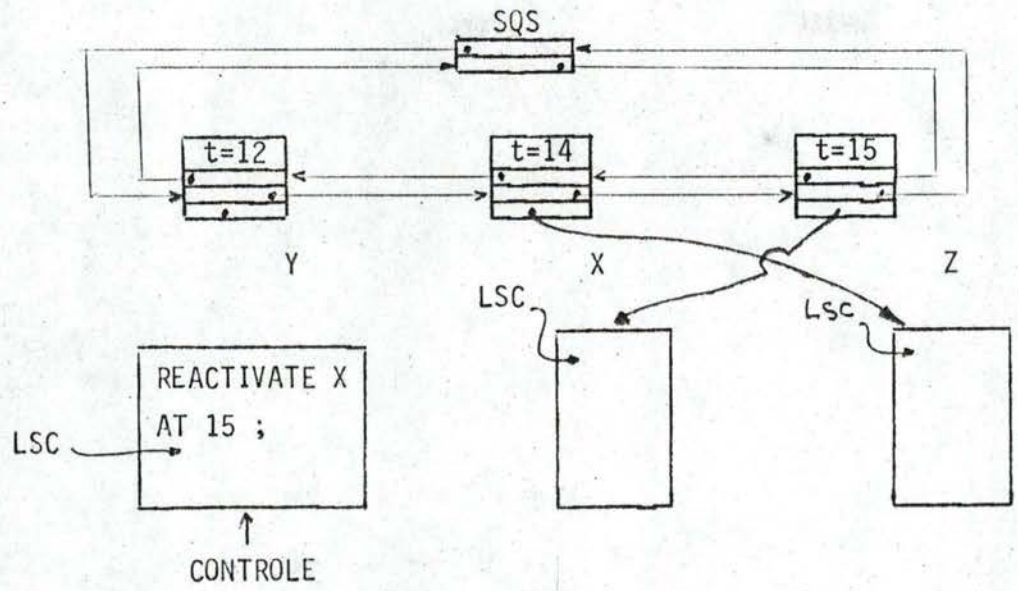
Enoncé REACTIVATE

Cette procédure est utilisée de la même façon que la procédure ACTIVATE et sert à céduler un objet d'une PROCESS CLASS *ayant déjà* une NE dans SQS.

REACTIVATE X enlève l'ancienne NE correspondant à X et en crée une nouvelle en temps voulu.

Exemple : REACTIVATE X AT 15





Enoncé HOLD

La procédure HOLD(T) enlève la NE de l'objet courant dans SQS et la replace à TIME+T. Le LSC ou pointeur de réactivation est placé à l'énoncé suivant le HOLD. L'énoncé représente une période inactive de durée T du processus courant.

Note : étant donné que le programme principal est lui-même un PROCESS, on pourra y utiliser HOLD. Entre autres, si on veut arrêter l'exécution du programme principal, simuler pendant un temps T, puis revenir au programme principal (par exemple pour l'impression des résultats), il suffit d'initialiser la simulation dans le programme principal et faire HOLD(T).

Enoncé PASSIVATE

Cette procédure utilisée dans le contexte quasi-parallèle est analogue à la procédure DETACH. Elle enlève la NE correspondant au PROCESS courant et place le pointeur de réactivation (LSC) à l'énoncé suivant son appel. Le contrôle de l'exécution passe donc au PROCESS dont la NE suit immédiatement dans SQS. Seul un ACTIVATE peut réintroduire dans SQS une NE correspondant à cet objet. On devra donc toujours prévoir un moyen de le retrouver, soit en le mettant dans une file, soit en lui attribuant soi-même un pointeur.

Enoncé WAIT(S)

où S est un pointeur à une file. WAIT(S) place l'objet courant dans la file S et fait un appel à la procédure PASSIVATE.

Enoncé CANCEL(X)

Cette procédure enlève la NE correspondant à X dans SQS. Si X était actif (objet courant), il devient passif.

3.5.3. Procédures utiles

IDLE

Cette procédure booléenne permet de vérifier si un PROCESS a une NE dans SQS. Ainsi IDLE prendra (par un appel de la forme PT.IDLE) la valeur Faux si le PROCESS correspondant au pointeur PT a une NE dans SQS.

EVTIME

Cette procédure, par un appel de la forme PT.EVTIME prend la valeur de l'heure indiquée dans la NE correspondant au PROCESS pointé par PT.

CURRENT

Cette procédure est un pointeur au process courant (dont la NE est en tête de file dans SQS).

Exemple : CANCEL (CURRENT) est équivalent à PASSIVATE
CANCEL CURRENT DELAY T est équivalent à HOLD(T).

NEXTEV

Cette procédure, par un appel de la forme PT.NEXTEV, fournit un pointeur au PROCESS dont la NE est située dans SQS, immédiatement après celle correspondant au PROCESS pointé par PT.

TIME

Cette procédure nous donne le temps courant depuis le début de la simulation.

Le tableau suivant nous donne toutes les procédures disponibles à l'utilisateur qui aurait préfixé une classe de son programme à l'aide de la classe SIMULATION.
Comme nous l'avons vu, cette classe SIMULATION est elle-même préfixée par la classe SIMSET (qui fournissait les procédures utiles au traitement de liste).

```
SIMSET  class  SIMULATION ;
        begin

LINK    class  EVENT NOTICE ; ... ;  cette classe permet d'ordonner les
                                     NE dans SQS.
LINK    class  PROCESS ;
        begin
                boolean procedure IDLE ;
                real procedure  EVTIME ;
                ref(PROCESS) procedure  NEXTEV ;
        end ;

        ref(PROCESS) procedure  CURRENT ;  ...  ;
        real procedure  TIME            ;  ...  ;
        procedure  HOLD ;               ;  ...  ;
        procedure  PASSIVATE ;          ;  ...  ;
        procedure  ACTIVATE ;           ;  ...  ;

        end  de la classe  SIMULATION ;
```

°

° °

CHAPITRE IV

GPSSS 5

4.1. INTRODUCTION

Ce chapitre décrit la *class* GPSSS5, écrite à l'Université de Montréal par Monsieur Jean Vaucher, professeur agrégé au Département d'Informatique.

GPSSS (General Purpose Simple / Super / Simulated Simulation System), qui n'est pas standard au compilateur SIMULA, est une extension à la classe SIMULATION et regroupe les concepts utiles pour la simulation de problèmes de files d'attente.

Donc, inspiré de l'idée des classes standards de SIMULA : SIMSET et SIMULATION, GPSSS5 a été écrit pour permettre à un usager de programmer certains modèles aussi facilement qu'avec la langage GPSS de Gordon.

Ceci est dû au fait que l'emploi de SIMULA comme langage de base permet les facilités de la programmation qu'offrent les langages de haut-niveau comme les structures de blocs, les procédures, les entrées-sorties ...

La philosophie de GPSS de Gordon a été gardée mais l'implantation n'a pas été suivie fidèlement. De même, le nom de certaines procédures (blocs) de GPSS a été changé.

G P S S S 5 contient les définitions de classes pour :

1. les transactions
2. les facilités
3. les storages
4. et les régions (files).

Le bloc ADVANCE de GPSS a été remplacé par les instructions plus flexibles de SIMULA : HOLD(T), ACTIVATE et REACTIVATE. De plus, une procédure WAIT UNTIL est disponible pour l'ordonnancement conditionnel.
On peut assigner des priorités aux transactions selon les politiques FIFO, LIFO et autres.
D'autres procédures nous donnent des renseignements quant au statut des ressources. L'utilisateur peut également demander l'impression de statistiques durant l'exécution et peut recommencer la simulation pour essayer d'autres politiques.

Nous allons maintenant expliquer la structure d'un programme GPSSSS5 et l'effet des procédures accessibles.

4.2. STRUCTURE DU PROGRAMME

GPSSS5 a été écrit comme étant une classe et contient environ 950 cartes.

En fait, l'utilisateur doit placer son programme (préfixé par GPSSS5) à la fin du programme source GPSSS5 et doit compiler et exécuter le tout. Le programme résultant a la forme suivante :

```
begin
        SIMULATION class  GPSSS5;
                  begin                          950 cartes
                  :
                  end ;
        GPSSS5    begin
                  :       USER PROGRAM
                  end ;
end ;
```

Le programme de l'usager (USER PROGRAM) se divise en deux parties :

1. Définition des attributs et actions des transactions
2. Le programme principal qui initialise et contrôle la simulation. C'est ici, par exemple qu'on doit créer le premier client.

4.2.1. Programme simple

Voici un programme GPSSS5 où des clients arrivent dans le système de manière aléatoire, y passent 10 minutes et sortent :

```
GPSSS5 begin

TRANSACTION class  CLIENT ;
           begin
                   ACTIVATE new CLIENT DELAY UNIFORM (0,10,u) ;
                   HOLD (10)
           end ;
      ACTIVATE new CLIENT DELAY 0 ;
      HOLD (100) ;                          Programme principal
      end ;
```

L'integer u sert de germe aléatoire à la procédure UNIFORM.

Le principal crée et active le premier client avec l'énoncé ACTIVATE de la même manière qu'une simulation utilisant la classe SIMULATION. Le programme principal s'arrête après 100 minutes de temps.

Les transactions sont définies comme des classes ordinaires mais préfixées par TRANSACTION.
Cette classe TRANSACTION *définie* dans GPSSS5, est elle-même préfixée par PROCESS (voir 4.2.2.).
Il est donc possible de se servir des énoncés d'ordonnancement de SIMULATION pour contrôler l'exécution d'objets de type TRANSACTION.
On remarque l'utilisation de l'énoncé ACTIVATE pour que chaque client génère son successeur. On indique que claque client passe 10 minutes dans le système avec HOLD(10).

4.2.2. Structure de la calsse GPSSS5

Donnons une structure simplifiée de la classe GPSSS que nous détaillerons au fil des sections :

```
SIMULATION class GPSSS5 ;
begin
      ref (HEAD) FACILITYQ, REGIONQ, STORAGEQ, TABLEQ ;
      integer TRANSID, PASS, U ;
      real TIME ORIGIN, SIMULATION START TIME ;
  LINK class ENTITY(IDENT) ; text IDENT ;
ENTITY class FACILITY ;                              begin ... end
ENTITY class STORAGE (CAPACITY) ;                    begin ... end
ENTITY class REGION ;                                begin ... end
ENTITY class GROUP (TAILLE) ;                        begin ... end
ENTITY class TABLE (N,LOWER, INTER) ;                begin ... end

PROCESS class TRANSACTION
begin
      real TIME MARK, PRIORITY, LAST REGION ENTRY TIME ;
      integer ID ;
      ref(REGION) LAST REGION ;
      procedure {ENTER, LEAVE} {FACILITY(F), STORAGE(S,-), REGION(R)} ;   begin ... end
      procedure  PRIORITY INTO(Q) ;                  begin ... end
      procedure  WAIT UNTIL(B) ;                     begin ... end
      procedure  JOIN(G) ;                           begin ... end
end ;
```

```
[ PROCESS class WAIT MONITOR ;

[ procédures utilitaires  SKIP(N), GPSSS TIME, CLEAR SQS, RESTART
                          RESET ;

  enquiry procédures      [CONTENTS]      [FACILITY(F)]
                          [WAITING ]      [STORAGE(S)]
                                          [REGION(R)]
                                          [GROUP(G)]
                                          [TABLE(T)]

  report generator        [FACILITIES]
  = STANDARD REPORT       [STORAGES ]     [REPORT]
                          [REGIONS  ]
                          [TABLE    ]

  error package
  = ERROR REPORT

  initialisation          [FACILITYQ]
                          [REGIONQ  ]
                          [TABLEQ   ]     : - new HEAD
                          [WAITQ    ]

                          WAIT MONITOR  : - new WAIT MONITEUR ;

  INNER

  ERROR REPORT

  end de la classe GPSSS5 ;
```

4.2.3. Correspondance avec GPSS de Gordon

GPSSS5 a été fortement inspiré de GPSS et on y retrouve des class FACILITY, STORAGE et REGION qui sont l'équivalent des stations simples, stations multiples et queues.

Ces classes représentent des objets passifs contenant seulement les variables nécessaires à l'accumulation de statistiques.

Les procédures pour la capture et libération des ressources sont placées dans une class TRANSACTION.

La liste suivante donne la correspondance entre ces procédures et les énoncés équivalents en GPSS :

GPSS	GPSSS5
SEIZE N	ENTER FACILITY (N)
RELEASE N	LEAVE FACILITY (N)
ENTER N, SIZE	ENTER STORAGE (N,SIZE)
LEAVE N, SIZE	LEAVE STORAGE (N,SIZE)
QUEUE N	ENTER REGION (N)
DEPART N	LEAVE REGION (N)

Dans les procédures GPSSS5, N représente une référence à la ressource ou région appropriée et dans le cas d'une storage, SIZE est un entier qui indique les unités de la ressource qui sont demandées ou rendues.

4.3. PROCEDURES POUR LES TRANSACTIONS

Toutes les classes préfixées par ENTITY (FACILITY, STORAGE, REGION, GROUP et TABLE) possèdent un paramètre IDENT.
Lors de la création des objets de ces différentes classes, une valeur initiale doit être spécifiée par le programmeur à ce paramètre.

Dans le cas de la classe FACILITY, par exemple FACILITYQ comprendra tous les objets de type facilités qui comprend entre autres l'identificateur de cette facilité, et cette file sera créée lors de *l'initialisation*.
Il en est de même pour les autres classes préfixées par ENTITY.

Chaque transaction a 3 principales variables qui sont prédéfinies dans la class GPSSS :

```
integer ID ;
real TIME MARK, PRIORITY ;
```

Quand une transaction est créée, PRIORITY est mis à zéro, TIME MARK au temps courant et un identificateur unique sera donné à l'entier ID.

PRIORITY est une variable permettant de fixer la discipline d'attente pour une FACILITY ou une STORAGE occupée. Normalement, les transactions ont accès aux ressources sur base FIFO mais si on change la valeur de PRIORITY, la transaction avec la plus petite sera servie en premier.

PRIORITY peut être négatif; dans ce cas, à valeur égale, la discipline est LIFO.

Nous verrons un exemple après avoir expliqué les différentes implantations des procédures pour les transactions.

4.3.1. Implantation des facilités

Avant de décrire les procédures ENTER et LEAVE pour les FACILITY; voyons d'abord la définition complète d'une FACILITY :

```
ENTITY class FACILITY
    begin
        ref(HEAD) INQ ;
        ref(TRANSACTION) OCCUPIER ;
        integer ENTRIES ;
        real TLAST, BUSY TIME ;

        INQ : new HEAD ;
        TLAST := TIME ORIGIN ;
        INTO(FACILITYQ) ;
    end ;
```

OCCUPIER désigne la transaction qui occupe la facilité; si cette facilité est libre la valeur de OCCUPIER est none.

INQ est une liste locale à chaque facilité où sont mises en attente les transactions qui essaient d'entrer dans la facilité lorsqu'elle est occupée.

ENTRIES compte le nombre de transactions qui ont occupé la facilité.

BUSY TIME accumule le temps d'utilisation de la facilité.

TLAST est une variable de travail dans laquelle on place l'heure à laquelle la facilité est devenue occupée.

Ces trois dernières variables servent à accumuler des statistiques.

Lors de l'initialisation du programme de l'utilisateur, nous aurons par exemple l'instruction :

GARAGE :- new FACILITY(""GARAGE FACILITY"") ;

où GARAGE FACILITY est l'identificateur de cette facilité créée, il sera placé dans FACILITYQ comme nous l'avions expliqué au début de la section 4.3.
Cette instruction aura en outre pour effet de créer une file INQ locale à cette facilité.
Le pointeur GARAGE contiendra l'adresse de cette facilité créée.

Voici maintenant la procédure ENTER FACILITY :

```
procédure ENTER FACILITY(F) : name F ; ref(FACILITY) F ;
begin
if F == none  then error ;
inspect F do begin
            if OCCUPIER == this TRANSACTION then ERROR
            else if OCCUPIER =/= none
                 then | begin
                      |
                      | PRIORITY INTO(INQ) ;
                 (2)  | PASSIVATE ;
                      | this TRANSACTION.OUT
                      | end
                 else | begin
                      | OCCUPIER :- this TRANSACTION ;
                 (1)  | TLAST := TIME ;
                      | end ;
end ENTER FACILITY ;
```

Une transaction fait appel à la forme (dans le programme de l'utilisateur)

ENTER FACILITY(GARAGE) ;

qui a pour effet de rechercher la facilité pointée par GARAGE. On procède aussi à une vérification logique pour détecter le blocage fatal occasionné par une transaction essayant d'entrer dans une ressource qu'elle occupe déjà.
Normalement, si la facilité est libre (1) on l'occupe et on note l'heure dans TLAST; sinon (2) la transaction est placée dans la file d'attente locale à cette facilité (INQ) à une place correspondant à sa priorité; de plus cette transaction

est passive.

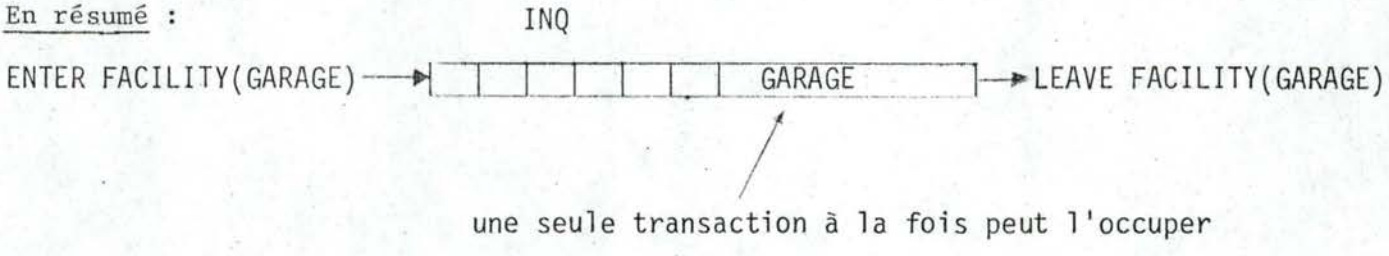
Voyons maintenant la procédure de sortie d'une facilité :

```
procedure LEAVE FACILITY(F) ; ref(FACILITY) F ;
begin
inspect F do if OCCUPIER =/= this TRANSACTION
              then ERROR
              else begin
                   OCCUPIER :- INQ.FIRST ;
                   if OCCUPIER == none
                   then BUSY TIME := BUSY TIME + TIME-TLAST
                   else ACTIVATE OCCUPIER DELAY 0
                   end
end LEAVE-FACILITY ;
```

Un appel de cette procédure pourrait être la forme LEAVE FACILITY(GARAGE); qui a pour effet :

- de vérifier d'abord si la transaction qui veut quitter la facilité est bien celle qui l'occupait.
- de parcourir INQ locale à cette facilité et regarder s'il y a d'autres transactions en attente pour rentrer dans la facilité, si oui on laisse rentrer la première de la file et on l'active sinon on accumule le temps d'utilisation.

En résumé :



4.3.2. Implantation des storages

Avant de décrire les procédures ENTER et LEAVE pour les STORAGES, voyons d'abord la définition complète d'une STORAGE.
Un storage a une capacité maximum(N) déterminée par le programmeur lors de l'initialisation de son programme. Un storage peut représenter un parking pour N voitures ou une mémoire d'ordinateurs à N mots. Une storage peut donc accomoder plus d'une transaction à la fois mais une fois pleine, elle refuse l'entrée aux transactions qui voudraient y rentrer.
Faisons l'hypothèse qu'une storage est totalement occupée et que les transactions en attente dans INQ demandent 4 unités, 2 unités et une unité de ressource respec-

tivement. Si a un moment donné, 3 unités sont libérées; ce n'est pas assez pour satisfaire la première requête mais assez pour les deux suivantes. GPSSS5 a choisi (comme GPSS) la solution de permettre aux 2 dernières requêtes de "depasser" la première.
(Procédure CHECK INQ employée lors d'un appel de sortie de la storage).

```
ENTITY class  STORAGE(CAPACITY) ; integer CAPACITY
      begin
            ref(HEAD) INQ ;
            integer MAX, ENTRIES, UNIT ENTRIES ;
            real CONTENTS, INTGRL, TLAST ;
            procedure CHECK INQ ;
               begin
               ref(TRANSACTION) CLIENT, NEXT CLIENT ;
               CLIENT :- INQ.FIRST ;
               while CLIENT =/= none and CONTENTS not equal
               CAPACITY do begin
                           NEXT CLIENT :- CLIENT.SUC ;
                           ACTIVATE CLIENT ;
                           CLIENT :- NEXT CLIENT ;
                           end
               end CHECK INQ ;
            INQ := TIME ORIGIN ;
            TLAST := TIME ORIGIN ;
            INTO(STORAGEQ) ;
      end ;
```

Lors de *l'initialisation* du programme de l'utilisateur, nous aurons par exemple l'instruction : PARKING :- new STORAGE (""voitures"", 50) ;
où 50 indique le nombre maximum d'emplacements disponibles dans ce parking.
Et de plus, supposons avoir 2 types de transactions voiture et camion.

La procédure d'entrée des transactions dans un storage :

```
procédure ENTER STORAGE(S,REQUIRED) ; name S; ref(STORAGE) S ;
                                      integer REQUIRED ;
begin
if S == none then
inspect S do if REQUIRED > CAPACITY then ERROR
          else begin
               PRIORITY INTO(INQ) ;
               while CONTENTS+REQUIRED > CAPACITY do PASSIVE
               this TRANSACTION.OUT ;
               ENTRIES := ENTRIES+1 ;
               UNIT-ENTRIES := UNIT ENTRIES + REQUIRED ;
               ACCUM(INTGRL, TLAST, CONTENTS, REQUIRED) ;
               IF CONTENTS > MAX then MAX := CONTENTS ;
               HOLD(0) ;
               end ;
end ENTER STORAGE ;
```

Un appel de cette procédure pourrait être de la forme ENTER STORAGE(PARKING, 2) ; qui signifie que nous voulons occuper 2 emplacements dans le parking.
Il faut vérifier si la demande d'unités de ressources de la storage n'est pas supérieure à la capacité maximum de cette storage.
Si toutefois, la somme du nombre d'unités demandées et présentement occupées venait à dépasser le nombre maximum d'unités possibles pour cette storage; la transaction qui en fait la demande sera passivée.
Une variable MAX contiendra en fin de simulation, le nombre maximum d'unités de ressources mobilisées par une transaction.

La procédure de sortie des transactions d'une storage est la suivante :

```
procédure LEAVE STORAGE(S,RELEASED); ref(STORAGE) S ;
                                     integer RELEASED ;
inspect S do begin
          ACCUM(INTGRL, TLAST, CONTENTS,-RELEASED) ;
          if CONTENTS   0  then begin
                               CONTENTS := 0 ;
                               ERROR ;
                               end ;
          CHECK INQ ;
          end LEAVE STORAGE ;
```

Un appel à cette procédure pourrait être de la forme suivante : LEAVE STORAGE(PARKING,1) : qui a pour effet de libérer un emplacement si par exemple notre transaction quittant la storage était une voiture.

La procédure ACCUM permet de recueillir diverses statistiques.

Une correction est à faire si une transaction libère plus d'unités que le nombre possible pour cette storage.

Si une transaction veut occuper un certain nombre d'unités de ressources et que cette demande est refusée; la procédure CHECK INQ réactivera les autres transactions de INQ si toutefois le nombre d'unités demandé ne dépasse pas le nombre disponible à ce moment.

4.3.3. Implantation des régions

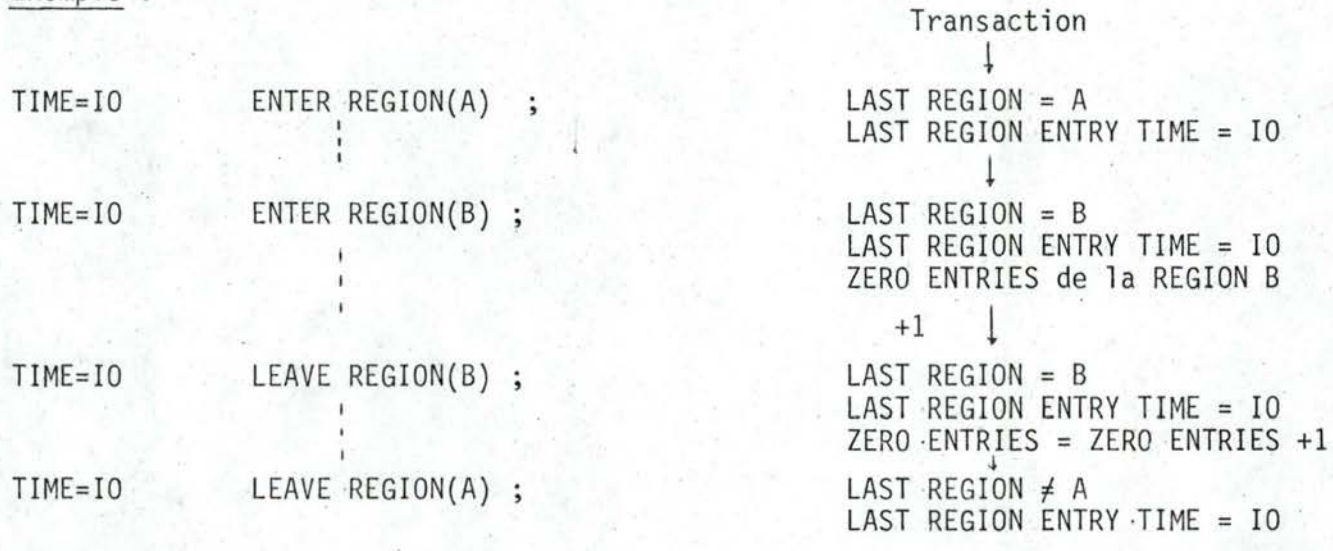
Une région a une capacité infinie. Donc une transaction ne se voit jamais refuser l'entrée. Une région, comme une queue de GPSS, sert simplement à mesurer le temps de passage dans une partie du programme et ce pour fins statistiques.

```
ENTITY class REGION :
       begin
              integer ENTRIES, ZERO ENTRIES, MAX;
              real    CONTENTS, INTGRL, TLAST;
              TLAST := TIME ORIGIN;
              INTO(REGIONQ);
       end;
procedure ENTER REGION(R) ; name R ; ref(REGION) R ;
begin
       if R == none then ERROR ;
       LAST REGION :- R ;
       LAST REGION ENTRY TIME := TIME ;
       inspect R do begin
                     ENTRIES := ENTRIES +1  ;
                     ACCUM(INTGRL,TLAST,CONTENTS,1) ;
                     if CONTENTS > MAX then MAX := CONTENTS
                     end ;
end ENTER REGION ;
procedure LEAVE REGION(R) : ref(REGION) R ;
inspect R do begin
              if LAST REGION == R and LAST REGION ENTRY TIME
```

```
                    = TIME
            then ZERO ENTRIES := ZERO ENTRIES + 1 ;
            ACCUM(INTGRL,TLAST,CONTENTS,-1) ;
            end ;
```

Toute transaction reçoit à sa création un réel LAST REGION ENTRY TIME et un pointeur à une région LAST REGION.
Toute région possède un entier ZERO ENTRIES qui nous donne le nombre de transactions qui n'ont pas subi de modifications de temps entre le moment d'entrée et de sortie d'une région.

Exemple :



vu que LAST REGION = B, la condition n'est pas vérifiée.

GPSS de Gordon permettait de recueillir les ZERO ENTRIES pour la région A, ce qui est impossible en SIMULA GPSSS5.

4.3.4. Implantation des tables

En GPSS, cela correspond aux blocs TABULATE et QTABLE. GPSSS nous donne la définition suivante de la classe TABLE qui permet au programmeur de recevoir en fin de simulation, un histogramme donnant une répartition de valeur d'une variable choisie.

```
ENTITY class TABLE(N,LOWER,INTER) ; integer N ;
                                    real LOWER, INTER ;
    begin
        integer array A [-1, N] ;
        integer ENTRIES ;
        real MIN, MAX, SUM ;
```

```
        procedure ADD(X) ; real X ;
        begin
        integer INDEX ;
        SUM := SUM + X ;
        ENTRIES := ENTRIES + 1 ;
        if ENTRIES = 1 then MIN := MAX := X
        else if X < MIN then MIN := X
        else if X > MAX then MAX := X ;
        INDEX := ENTIER((X-LOWER)/INTER) ;
        if INDEX < 0 then A[ -1] := A [ -1] + 1
        else if INDEX > N then A[ N] := A[ N] +1 ;
        else A[INDEX] := A[INDEX] +1 ;
        end ;
    INTO(TABLEQ) ;
    end TABLE DEFINITION ;
```

A la création de la table, on fournit dans l'ordre les paramètres suivants un nom pour l'impression, le nombre d'intervalles (N) pour l'histogramme, la borne supérieure du premier intervalle (LOWER) et la taille des intervalles (INTER).

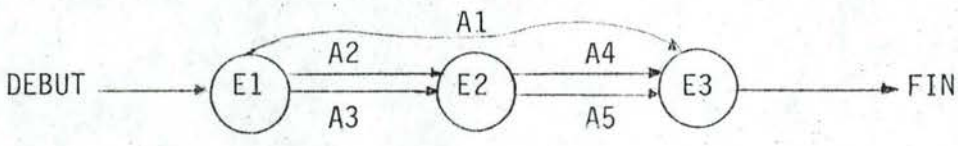
Exemple : DIST :- new TABLE(""TEMPS DANS EPICERIE"",10,50,50) ;
Dans le programme de l'utilisateur, on trouvera une instruction de la forme :

```
DIST.ADD(TIME-HEURE ARRIVEE CLIENT) ;
```

4.3.5. Implantation des groupes et l'opération JOIN

La notion de GROUP est surtout utile pour les problèmes de synchronisation comme les modèles de réseaux PERT où le déclenchement d'une activité dépend de la terminaison d'activités parallèles préalables.

Le graphe suivant montre la relation entre 5 activités nécessaires à la réalisation d'un projet :



Les arcs A1-A5 dénotent les activités et les noeuds E1-E2-E3 représentent les étapes dans la réalisation du projet. Quand toutes les activités qui aboutissent à un noeud sont terminées, l'étape est franchie et les activités qui sortent du noeud peuvent démarrer.

Dans notre exemple, au début du projet, on démarre les activités A1 A2 et A3. Les activités A4 et A5 doivent attendre que A1 et A2 soient toutes 2 terminées

pour partir.
Quand A1,A4 et A5 auront terminé, on dira que le projet est terminé.

Notre but est d'implanter correctement l'ordonnancement afin de trouver la durée.
Nous allons maintenant décrire le concept de GROUP et l'opération JOIN implantés dans GPSSS pour réaliser un tel schéma de synchronisation. Soulignons que GPSS contient 4 énoncés spéciaux : SPLIT ASSEMBLE GATHER et MATCH qui permettent les mêmes opérations.

Pour GPSSS5, le GROUP contient diverses transactions tandis qu'en GPSS de Gordon, il ne faut pas oublier la notion de famille de transactions.

Voici la définition de la classe GROUP

```
class GROUP(TAILLE) ; integer TAILLE ;
begin
      integer N ;
      ref(HEAD) INQ ;
      boolean procedure READY ;
          READY := (N+1 = TAILLE)
      INQ :- new HEAD ;
end GROUP DEFINITION ;
```

Cette classe représente la formation d'un groupe d'une taille spécifiée. Les transactions qui se joignent au groupe sont mises en attente dans INQ. La procédure READY permettra de vérifier que la prochaine arrivée déclenchera la libération. Pour qu'une transaction fasse partie d'un groupe, elle devra appeler la procédure JOIN implantée par GPSSS dans la classe TRANSACTION.

```
procedure JOIN(G) ; ref(GROUP) G;
begin
        ref(PROCESS) P ;
        inspect G do begin
                    N := N + 1 ;
                    if N<TAILLE
                    then begin
                         WAIT(INQ) ;
                         OUT ;
                         end
                     else begin
                          N := 0 ; P :- INQ.FIRST ;
```

```
                    while P =/= none do begin
                      ACTIVATE P DELAY 0;
                      P :- P.SUC ;
                                        end ;
                    HOLD(0) ;
                    end ;
                end ;
end PROCEDURE JOIN ;
```

4.4. PROCEDURES UTILITAIRES

Procedure SKIP(N) ; cette procédure de sortie des résultats sur imprimante permet de sauter N lignes d'impression après avoir fait un énoncé OUTIMAGE.

Ex. :
```
OUTTEXT ("EXEMPLE") ;
SKIP(2) ;
OUTTEXT ("FIN") ;
```
avec pour effet d'imprimer :

```
EXEMPLE
FIN
```

real procedure GPSSS TIME ; cette procédure nous donne le temps écoulé depuis le début de la simulation.

```
GPSSS TIME := TIME-SIMULATION START TIME ;
```

procedure RESTART ; cette procédure doit obligatoirement apparaître dans le programme principal et a pour effet d'imprimer le rapport d'erreurs ainsi que les statistiques de la simulation. De plus, elle refait une initialisation des différentes files de GPSSS pour pouvoir recommencer une nouvelle simulation. La procédure CLEAR remplit les mêmes fonctions en GPSS.

Ex.
```
  .
  .
init de l'user program
  .
  .
HOLD(60*60) ;
RESTART ;    statistiques de la première simulation
HOLD(8*60*60) ;
STANDARD REPORT  statistiques de la seconde simulation
end ;
```

Le RESTART fait un STANDARD REPORT implicite, mais en général le programmeur doit indiquer par l'appel de STANDARD REPORT la demande de statistiques.

procedure RESET ; cette procédure sert à initialiser les statistiques sans imprimer de rapport ni changer l'état du système. Cela permet, ce qu'on appelle, un rechauffement de la simulation.

Ex. : HOLD(60*60) ; rechauffement d'une heure
RESET ;
HOLD(8*60*60) simulation pendant 8 heures.

Nous aurons donc des statistiques seulement pour les 8 dernières heures de la simulation.

procedure CLEAR SQS ; cette procédure sert à vider la file SQS contenant les notices d'événements. Elle revient donc à arrêter la simulation.

Ex. : WHILE CURRENT.NEXTEV =/= NONE DO CANCEL (CURRENT.NEXTEV) ;

4.5. PROCEDURES GENERALES

CONTENTS FACILITY(F)
WAITING STORAGE(S) ;
REGION(R)
GROUP(G)
TABLE(T)

- La procédure WAITING utilisée principalement pour les FACILITY et STORAGE nous donne le nombre de transactions attendant dans INQ que la ressource soit libérée.
  Cette procédure retournera la valeur 0 dans le cas d'une région car une transaction ne se voit pas refuser l'entrée dans une région.
- La procédure CONTENTS nous donne le contenu d'une ressource maximum de CAPACITE et pour les FACILITY, STORAGE et REGION respectivement.
- La procédure STANDARD REPORT nous donne à la fin de la simulation les différentes statistiques accumulées durant la simulation.

°

° °

CHAPITRE V

ETUDE DE NOUVELLES PROCÉDURES

5.0. INTRODUCTION

L'extension GPSSS5 de la classe SIMULATION, écrite par Monsieur J. Vaucher, professeur agrégé à l'Université de Montréal, a été convertie et précompilée pour devenir opérationnelle sur l'ordinateur DEC 2050 de l'Institut d'Informatique de Namur.

L'apport de nouvelles procédures, intégrées à l'extension GPSSS5 de la classe SIMULATION, pouvait encore faciliter la programmation et l'analyse des résultats de simulations.

Le choix s'est porté sur l'analyse des files d'attente propres à chaque ressource (FACILITY ou STORAGE) et plus particulièrement au nombre et au temps moyen des transactions en attente de cette ressource.

La possiblité est offerte aussi au programmeur d'éditer un rapport décrivant le cheminement d'un certain nombre de transactions parmi les ressources qu'elles ont occupées.

5.1. STATISTIQUES RELATIVES AUX FILES D'ATTENTE

5.1.1. Introduction

Nous avons vu que l'appel à la procédure STANDARD REPORT nous fournissait, en fin de simulation, des renseignements relatifs aux FACILITY, STORAGE et REGION définis dans le programme de l'utilisateur.

Aucune information statistique n'est donnée quant aux files d'attente propres à chaque FACILITY et STORAGE.
Cette file d'attente INQ contient la liste d'adresses des transactions en attente d'occuper la ressource.

De ce fait, une amélioration de la classe GPSSS5 peut être l'analyse de ces files d'attente INQ et plus particulièrement le nombre et le temps moyen des transactions en attente d'occuper la FACILITY ou STORAGE.
De plus, un intervalle de confiance sera calculé pour chacune de ces moyennes.

5.1.2. Collecte d'observations

La collecte d'observations de type *temps moyen d'attente* se fait de la manière suivante.
A chaque fois qu'une transaction entre dans une FACILITY ou STORAGE (c'est-à-dire mise de l'adresse de cette transaction dans la liste INQ) on mémorise ce temps d'entrée. Lorsque, plus tard, cette transaction quittera la file d'attente INQ pour occuper la ressource devenue libre, on pourra faire la différence de ce temps et le temps d'entrée.

Ce résultat constituera une observation du temps d'attente et sera placé dans une liste (LISTE2) propre à cette FACILITY ou STORAGE dans laquelle la transaction a pénétré.
Lors de l'édition du STANDARD REPORT en fin de simulation, le temps moyen d'attente sera le quotient de la somme des observations par leur nombre.

La collecte d'observations de type *nombre moyen de transactions en attente* peut se faire de deux manières différentes :

Première manière

Le programmeur a fait un appel à la procédure OBSERVATION AUTOMATIQUE. (Cette procédure sera détaillée plus loin dans ce chapitre). Dès lors, la collecte d'observations se fera à des temps fixés par le programmeur et ce, pour toute la durée de la simulation.

Deuxième manière

Si au contraire, l'appel de la procédure OBSERVATION AUTOMATIQUE n'a pas été spécifié dans le programme de l'utilisateur, la collecte se fera lors de tout appel d'une des procédures :

$$\begin{bmatrix}\text{ENTER}\\ \text{LEAVE}\end{bmatrix} \quad \begin{bmatrix}\text{FACILITY}\\ \text{STORAGE}\end{bmatrix}$$

Cela revient à dire que lorsqu'une transaction veut occuper ou libérer une ressource, une prise d'observation de type WAITING FACILITY(name) ou WAITING STORAGE(name) nous donnant le nombre de transactions en attente (c'est-à-dire la cardinalité de la liste INQ), sera effectuée.

Ayant choisi l'une ou l'autre manière, le nombre moyen de transactions en attente, à la fin de la simulation sera toujours le quotient de la somme des observations par leur nombre. Ces observations ayant été placées dans une liste (LISTE) propre à chaque FACILITY ou STORAGE.

Voyons maintenant l'implémentation de la collecte d'observations pour les FACILITY.

```
• SIMULATION class GPSSS5 ;
begin
            ...
            ref(HEAD) FACILITYQ ;
            ...
LINK class  OBS (X) ; integer X;
LINK class  ENTITY (IDENT) ; value IDENT ; text IDENT ;

• ENTITY class FACILITY ;
begin
       ref(HEAD) INQ ;
       ref(TRANSACTION) OCCUPIER ;
       ref(HEAD) LISTE, LISTE2 ;
   LISTE :- new HEAD ;
   LISTE2 :- new HEAD ;
   ...
   INTO (FACILITYQ)
end facility definition ;

• PROCESS class TRANSACTION;
begin
       real TEMPS ENTREE ;
procedure ENTER FACILITY (F); name F ; ref (FACILITY) f ;
   begin
   if F == none  then f :- new FACILITY ("ERR") ;
   inspect F do
      begin
      if not COLLECT AUTOM  then new OBS (WAITING FACILITY(F))
      .                                    .INTO (LISTE) ;
      :
      TEMPS ENTREE := TIME ;
      if OCCUPIER == this TRANSACTION then ERREUR
      else if OCCUPIER =/= none then
              begin
                PRIORITY INTO (INQ) ;
                PASSIVATE ;
                this TRANSACTION.OUT ;
              end
                           else
             OCCUPIER :- this TRANSACTION ;
      ENTRIES := ENTRIES + 1 ;
      new OBS (TIME - TEMPS ENTREE).INTO (LISTE2) ;
      end ;
   end procedure ENTER FACILITY ;
```

```
    procedure LEAVE FACILITY (F) ; ref (FACILITY) F ;
      begin
      inspect F do begin
          if not COLLECT AUTOM  then new OBS (WAITING FACILITY(F))
                                                .INTO (LISTE) ;
          if OCCUPIER =/= this TRANSACTION then ERREUR
          else begin
               OCCUPIER :- INQ.FIRST ;
               if OCCUPIER =/= none then
                      ACTIVATE OCCUPIER DELAY 0 ;
          end
                      end
      otherwise ERREUR ;
    end procedure LEAVE FACILITY ;
     :
    end TRANSACTION definition ;

                      :

    procedure STANDARD REPORT ;
      begin
        :
      end ;

INNER ;

ERROR REPORT ;

end GPSSS5 definition ;
```

5.1.3. Intervalle de confiance

Un calcul d'intervalle de confiance pour le temps et le nombre moyen de transactions en attente a été implémenté.
Lors de l'édition du STANDARD REPORT, ils y figureront parmi les autres statistiques.

Voyons maintenant la construction de l'intervalle de confiance décrite dans la classe INTERVALLE.

Ayant recueillis n observations, on peut employer deux méthodes différentes dont chacune comprend soit une solution de Bienaymé-Tchbychev, soit une solution normale.
Nous verrons plus loin les conditions pour employer telle méthode et telle solution.

Soit $MOY = (\sum_{i=1}^{n} x_i)/n$ la moyenne des observations recueillies, qui est un estimateur non biaisé de la moyenne M dont on ne veut un calcul d'intervalle de confiance.

PREMIERE METHODE

On calcule la somme des carrés des observations pour obtenir la variance : $\frac{1}{n} \sum_{i=1}^{n} (x_i)^2 - Moy^2$

L'écart-type σ_{MOY} est la racine carrée de la variance, nous pouvons en déduire l'intervalle de confiance par l'une des solutions suivantes :

Solution Beinaymé

Nous avons, grâce à l'inégalité de Bienaymé-Tchebychev :

$\forall \alpha > 0$: Prob $(|Moy - M)| \geq \alpha) \quad (\frac{\sigma\ MOY}{\alpha})^2$

ou encore en posant $\alpha = \delta\sigma_{MOY}$

$\forall \delta > 0$: Prob $(|MOY - M| \geq \delta\sigma_{MOY}) \leq \frac{1}{\delta^2}$

$\forall \delta > 0$: Prob $(|MOY - M| < \delta\ \sigma_{MOY}) \geq 1 - \frac{1}{\delta^2}$

En choisissant de calculer l'intervalle de confiance avec une probabilité au moins égale à 0.95, nous posons donc :

$1 - 1/\delta^2 = 0.95$, c'est-à-dire $\delta = \sqrt{20} = 4.47$

Nous pouvons donc affirmer que :

$MOY - 4.47 * \sigma_{MOY}/\sqrt{n} < M < MOY + 4.47 * \sigma_{MOY}/\sqrt{n}$ avec une probabilité au moins égale à 0.95.

Solution normale

Le quantile de GAUSS à l'intervalle de confiance au niveau 5% étant 1.96, nous pouvons affirmer que :

$MOY - 1.96 * \sigma_{MOY}/\sqrt{n} < M < MOY + 1.96 * \sigma_{MOY}/\sqrt{n}$ avec une probabilité au moins égale à 0.95.

DEUXIEME METHODE

Nous pouvons prendre comme solution pour la calcul de l'intervalle :

soit $MOY - \frac{4.47}{1.96} \sigma_{MOY} < M < MOY + \frac{4.47}{1.96} \sigma_{MOY}$

Pour estimer l'écart-type σ_{MOY}, une méthode consiste à fractionner l'échantillon en p blocs d'observations de nombre fixe :

$(x_1\ x_2 \cdots x_k)\ (X_{k+1} \cdots X_{2k})\ (X_{2k+1} \cdots X_{3k}) \cdots (X_{(p-1)k+1} \cdots X_n)$.

bloc 1 bloc 2 bloc 3 bloc p

On choisit par tatonnements successifs et pour tous les blocs une même taille k de telle façon que l'auto-corrélation entre les extrêmes de chaque bloc soit, en valeur absolue, nettement inférieure à 1 :

$$\frac{\nu(k)}{\nu(0)} \gg 1$$

Pour ce faire, on a recours à l'estimateur :

$$\nu(n) = 1/_{n-h} \sum_{i=1}^{n-h} (X_i - MOY) * (X_{i+h} - MOY).$$

Il s'agit de calculer ensuite la moyenne pour chaque bloc j (j=1,2,...p) :

$X^{(j)} = 1/k \sum_{i=1}^{k} X_{q+i}$ avec q = (j-1)k, de telle façon que l'on puisse considérer $(X^{(1)} \cdots X^{(2)} \cdots X^{(p)})$ comme un échantillon non auto-corrélé de la variable aléatoire MOY.

Voyons maintemant les deux solutions possibles pour cette méthode.

Calculons la moyenne $\overline{X} = \Sigma X^{(j)}/p$

$[\sum_{j=1}^{p} (X^j)^2 - (\sum_{j=1}^{p} X^{(j)2}/p\ ]\ /p-1$ est un estimateur non biaisé de la variance des moyennes de blocs.

Solution Bienaymé

Nous avons donc comme intervalle de confiance :

$$\bar{x} - 4.47\ \sigma_{\bar{x}} < M < \bar{x} + 4.47\ \sigma_{\bar{x}}$$

Solution normale

$$\bar{x} - 1.96\ \sigma_{\bar{x}} < M < \bar{X} + 1.96\ \sigma_{\bar{x}}$$

Nous verrons plus loin (voir 5.2.) qu'il y aura moyen d'arrêter la simulation lorsque la longueur de l'intervalle calculé est >= à la longueur désirée par le programmeur. Cette longueur est représentée par tout ce qui se trouve à droite de MOY $\pm$ et qui portera le nom de SOMME dans la classe INTERVALLE.

CHOIX DE LA METHODE

Nous sommes d'accord d'envisager la solution normale lorsque le nombre d'observations (Méthode 1) ou le nombre de blocs (Méthode 2) dépasse 30 unités.
Si le nombre de blocs était inférieur à 4 unités la méthode des blocs (Méthode 2) ne donnerait pas un sens véritable à l'écart-type et il faudrait calculer l'intervalle de confiance par la 1ère méthode.
Nous pouvons donc en déduire :

soit n le nombre d'observations
soit nb le nombre de blocs.

nb ⩽ 4	n ⩽ 30	solution Bienaymé	Méthode 1
	n > 30	solution Normale	Méthode 1
4 < bn ⩽ 30		solution Bienaymé	Méthode 2
nb > 30		solution Normale	Méthode 2

Nous trouverons en annexe l'implémentation de la classe INTERVALLE.

5.2. CALCUL D'INTERVALLES DE CONFIANCE EN COURS DE SIMULATION

5.2.1. Introduction

Nous avons vu que le calcul d'intervalles de confiance pour le temps et le nombre moyen de transactions en attente d'une ressource était réalisé, à la fin de la simulation lors de l'appel à la procédure STANDARD REPORT.

Une procédure INTER CONFIANCE peut être appelée dans le programme de l'utilisateur (avant le premier ACTIVATE exécutable) afin de tester, pour une ressource bien définie et à chaque nouvelle observation, l'intervalle de confiance. Si le test, portant sur la longueur désirée par le programmeur de l'intervalle de confiance, est réalisé; il s'agira d'arrêter la simulation. Nous voyons, ici, la nécessité d'implémenter le calcul de l'intervalle de confiance en une classe (INTERVALLE) et non en une procédure car son appel se fera à chaque nouvelle observation.

La procédure INTER CONFIANCE comprend les paramètres suivants :

- nom de l'entité ou ressource dont on veut tester la condition sur l'intervalle de confiance,
- la valeur du rapport NUM/DENOM qui par défaut est à .2 (voir 5.1.3.),
- la longueur admise ou désirée pour l'intervalle de confiance,
- le type d'observations :
  0 = nombre moyen de transactions en attente.
  1 = temps moyen d'attente des transactions.

5.2.2. Implémentation de la procédure INTER CONFIANCE

La simulation pourra être arrêtée lorsque la longueur réelle (calculée dans la classe INTERVALLE) de l'intervalle de confiance est >= à la longueur admise ou désirée par le programmeur. Cette condition est vérifiée à la fin de la classe INTERVALLE. Après arrêt de la simulation, une série de statistiques sont données pour cette ressource dont on a testé la longueur de l'intervalle de confiance.

Voyons l'implémentation de la procédure INTER CONFIANCE et la condition d'arrêt testée dans la classe INTERVALLE :

```
process class INTERVALLE (NB,FILE) ;   ref (HEAD) FILE ;
                                        integer NB ;
```

```
PROCESS class INTERVALLE (NB,FILE) ;  ref (HEAD) FILE ;
                                      integer NB ;
begin
      :
    calcul de SOMME = longueur réelle de l'intervalle de confiance.
if ARRET SIMULATION
                  then begin impression d'une ligne de report
 if SOMME > = LONG ADM then begin
              impression du final du report.
                   CLEAR SQS
                   ARRET SIMULATION := false ;
                   REACTIVATE MAIN ;
                        end ;
                     end ;
end procédure INTERVALLE
 :
procedure OBSERVATION AUTOMATIQUE ; begin  ...  end ;
integer I C TYP ;
real PARAM , LONG ADM ;          déclarations au niveau GPSSS5
ref (ENTITY) I C NOM ;
boolean ARRET SIMULATION ;

procedure INTER CONFIANCE (ENTITY NOM, RAPPORT, LONGUEUR, TYPE) ;
        name ENTITY NOM ;
        ref (ENTITY) ENTITY NOM ; real RAPPORT ;
        real LONGUEUR ;  integer TYPE ;
begin
  I C NOM :- ENTITY NOM ;
  PARAM := RAPPORT ;
  LONG ADM := LONGUEUR ;
  I C TYP := TYPE ;
  ARRET SIMULATION := true ;
  impression de l'entête du report.
end procédure INTER CONFIANCE ;
```

La classe INTERVALLE sera activée lors de toute nouvelle observation de type défini par le programmeur, de la ressource. Voyons l'implémentation de cette instruction d'activation dans la procédure ENTER FACILITY :

```
procedure ENTER FACILITY (F) ; ... ;
begin
   if F == none then ERREUR ;
   inspect F do begin
        if not COLLECT AUTOM then begin
            new OBS (WAITING FACILITY (F)).INTO(LISTE) ;
            if I C NOM == F and I C TYP = 0
                          and ARRET SIMULATION then
            ACTIVATE new INTERVALLE (LISTE, LISTE.CARDINAL) ;
                                        end ;
:
end ENTER FACILITY ;
```

La signification du booléen COLLECT AUTOM est explicite dans le paragraphe suivant.

L'appel à la procédure INTER CONFIANCE nous imprime les éléments suivants :

- à chaque nouvelle observation :

  si loi BIENAYME

  nombre d'observations, nombre de blocs et la longueur calculée de l'intervalle.

  si loi NORMALE

  nombre d'observations et longueur de l'intervalle.

- à l'arrêt de la simulation :

  variance, écart-type et longueur de l'intervalle de confiance.

5.3. COLLECTE AUTOMATIQUE D'OBSERVATIONS

5.3.1. Introduction

Le programmeur peut réaliser une collecte automatique d'observations par un appel à la procédure OBSERVATION AUTOMATIQUE.

Cette procédure a pour rôle de collecter, tout au long d'une simulation, et à des moments fixés par le programmeur, un certain nombre d'observations.

Elle aura pour seule paramètre le nombre d'unités de temps s'écoulant entre deux collectes d'informations.
Le type d'observation est *le nombre de transaxtions en attente* de toute FACILITY ou STORAGE créée par le programme de l'utilisateur.

Ces différentes observations seront exploitées par la procédure STANDARD REPORT à la fin de la simulation (1).

Voyons maintenant l'implémentation de cette procédure dans la classe GPSSS5 de SIMULA.

5.3.2. Localisation dans la classe GPSSS5

```
SIMULATION class GPSSS5 ;
begin
    - déclarations -
                    FACILITY
                    STORAGE
    ENTITY  class   REGION      ;       begin ... end ;
                    GROUP
                    TABLE
    LINK    class   OBS (X) ;
    PROCESS class   TRANSACTION ;       begin ... end ;
    PROCESS class   OBSERVATION ;       begin ... end ;
    PROCESS class   WAIT-MONITOR ;      begin ... end ;
    - procédures utilitaires -
    - enquiry procedures -
    procedure STANDARD REPORT ;         begin ... end ;
    procedure OBSERVATION AUTOMATIQUE ;
                                        begin ... end ;
    procedure ERROR PACKAGE ;           begin ... end ;
    - initialisation -
    INNER ;
    ERROR REPORT ;
end class GPSSS5 ;
```

(1) fin normale ou fin forcée déclanchée par une condition sur un intervalle de confiance si toutefois le programmeur avait fait appel à la procédure INTER CONFIANCE (voir 5.2.).

5.3.3. Implémentation de la procédure

```
SIMULATION class GPSSS5 ;
begin
        :
      integer TIME BTW OBS ;
        :
LINK class OBS (X) ; value X ; integer X ;
  :
PROCESS class OBSERVATION ;
begin
      ref (ENTITY) P ;
ACTIVATE new OBSERVATION DELAY  TIME BTW OBS ;
for P :- FACILITYQ.FIRST, STORAGEQ.FIRST do
while P =/= none do begin
   inspect P when FACILITY do begin
      new OBS (WAITING FACILITY(P)). INTO(LISTE) ;
      if I C NOM == F and I C TYP = 0
                    and ARRET SIMULATION
      then ACTIVATE new INTERVALLE (LISTE,LISTE.CARDINAL) ;
                                  end
   inspect P when STORAGE do begin
      new OBS (WAITING STORAGE(P)).INTO (LISTE) ;
            - idem que pour les facility -
                                  end ;
   P :- P.SUC ;
end class OBSERVATION ;
   :
 boolean COLLECT AUTOM ;
   :
```

```
procedure OBSERVATION AUTOMATIQUE (FREQUENCE) ;
                    integer FREQUENCE ;
begin
      COLLECT AUTOM := true ;
      TIME BTW OBS := FREQUENCE ;
      ACTIVATE new OBSERVATION DELAY 0 ;
end procédure observation automatique ;
   :
   :
end class GPSSS5 ;
```

L'appel à la procédure OBSERVATION AUTOMATIQUE, appel se trouvant dnas le programme de l'utilisateur avant le premier ACTIVATE exécutable,provoquera l'activation de la classe OBSERVATION toutes les "fréquences" unités de temps.

Chaque observation, comme dans la section précédente, sera placée dans une liste (LISTE). Cette liste existe lors de la création de toute FACILITY ou STORAGE dans le programme de l'utilisateur.

Rappelons que la procédure WAITING FACILITY (P) nous donne le nombre de transactions en attente de la FACILITY P.

5.4. CHEMINEMENT DES TRANSACTIONS

5.4.1. Introduction

Le programmeur peut faire le suivi d'un certain nombre de transactions par un appel à la procédure TRACE.

Un rapport édité montrera le chemin parcouru par la transaction dans le temps et à travers les différentes FACILITY et STORAGE.

L'appel à cette procédure doit se trouver avant le premier ACTIVATE exécutable du programme de l'utilisateur.

Le seul paramètre de cette procédure est le nombre de transactions dont on en veut la trace.

Une variable locale à chaque class TRANSACTION et nommée MARK TRACE peut permettre au programmeur de différencier les transactions créées dans les différentes classes de son programme.

Exemple d'un programme utilisateur

```
GPSSS5 begin
        :
TRANSACTION class VOITURE ;
begin
     MARK TRACE := 'V' ;
     ACTIVATE new VOITURE DELAY 3 ;
        :
end classe VOITURE ;

TRANSACTION class CAMION ;
begin
     MARK TRACE := 'C' ;
     ACTIVATE new CAMION DELAY 10 ;
        :
end classe CAMION ;
   :
- initialisation -
 TRACE (3) ;
 ACTIVATE new VOITURE DELAY 0 ;
 ACTIVATE new CAMION  DELAY 0 ;
   :
end GPSSS5 ;
```

Le résultat de l'appel TRACE (3) dans le programme utilisateur sera l'édition du rapport en fin de simulation .

TRACE REPORT

TRANSACTION 1 type is V

entered	PARKING	at time 0
entered	PLACE	at time 2
leaved	PLACE	at time 3
leaved	PARKING	at time 4

TRANSACTION 2 type is V

entered	PARKING	at time 0
entered	PLACE	at time 1
leaved	PLACE	at time 5
leavec	PARKING	at time 7

TRANSACTION 3 type is V

entered	PARKING	at time 3
entered	PLACE	at time 4
leaved	PLACE	at time 10
leaved	PARKING	at time 11

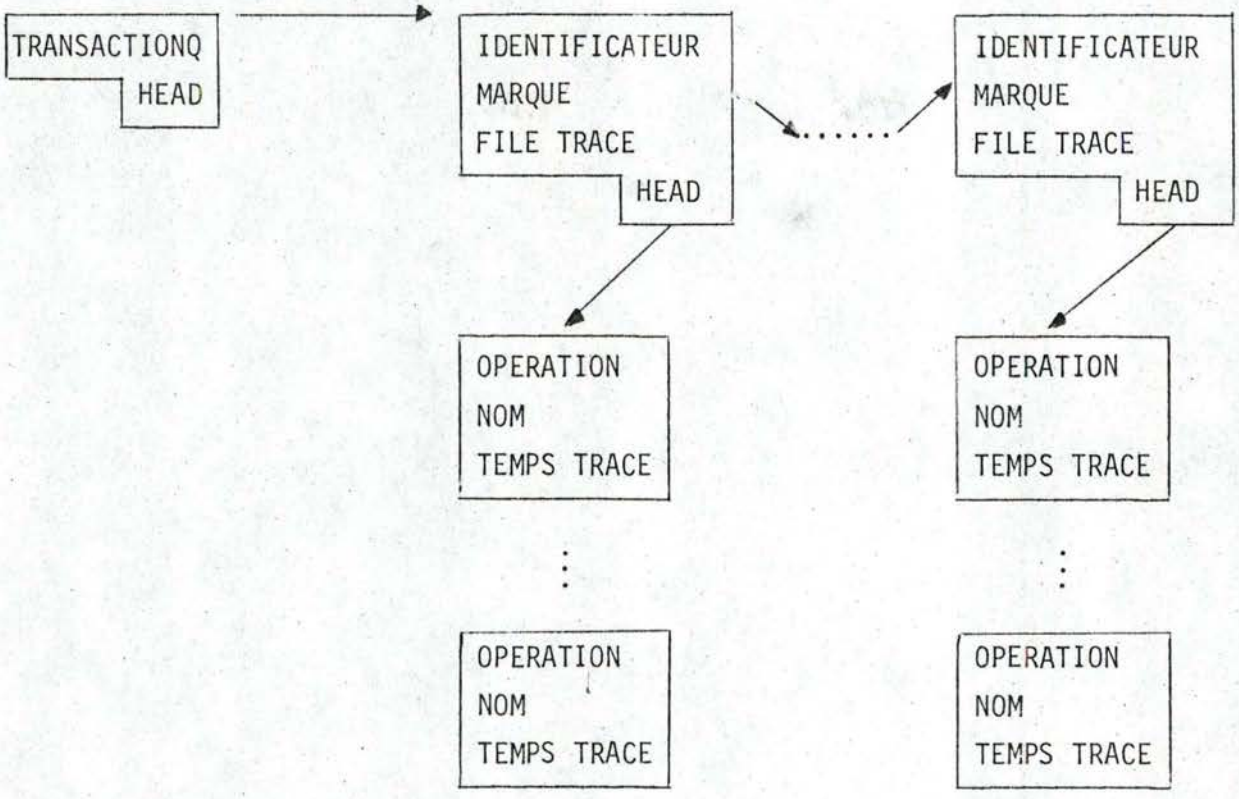
5.4.2. Définition de la procédure

Une file TRANSACTIONQ comprendra un descripteur par TRANSACTION créée dont on veut une trace pour toute la durée de la simulation.

Ce descripteur sera créé lors de l'activation de toute nouvelle transaction dont on en demande la trace.
Il comprendra l'identificateur de la transaction (ID), une zone MARK qui recevra le MARK TRACE (si toutefois une valeur lui a été assignée dans le programme de l'utilisateur, sinon elle sera laissée à blanc) et un pointeur à une file FILE TRACE qui contiendra les occurrences du chemin parcouru par cette transaction.

Cette file FILE TRACE propre à chaque transaction créée sera composée d'objets comprenant les éléments suivants :

- le type d'opération (ENTER ou LEAVE) ,
- l'identificateur de la FACILITY ou STORAGE,
- le temps (TIME) auquel cet événement se produit.



Le nombre d'éléments de la file TRANSACTIONQ sera donc le nombre de transactions dont on en veut la trace. Vu que les différentes files FILE TRACE se remplissent non-séquentiellement, il faudra, à chaque événement reconnaître la file où ranger l'information; cela se réalise grâce à l'identificateur défini dans le descripteur (procédure SEARCH FILE TRICE dans la classe TRANSACTION).

Les éléments de la TRANSACTIONQ seront définis comme objet de la classe PT-TRACE tandis que les éléments de la FILE TRACE seront définis comme objet de la classe TRANS.

5.4.3. Localisation dans la classe GPSSS5

```
SIMULATION class GPSSS5 ;
begin
        - déclarations -
LINK class OBS ;                  begin ...   end ;
LINK class PT-TRACE (.) ;         begin ...   end ;
LINK class TRANS (.,.,.) ;        begin ...   end ;
PROCESS class TRANSACTIONS ;      begin ...   end ;
```

```
procedure  STANDART REPORT ;        begin ... end ;
procedure TRACE (.) ;               begin ... end ;
        - initialisation -
INNER ;
ERROR REPORT ;
end classe GPSSS5 ;
```

5.4.4. Implémentation de la procédure TRACE

```
SIMULATION class GPSSS5 ;
begin
      ref (HEAD) TRANSACTIONQ ;
      boolean INIT TRACE, TRACE GEN ;
      integer MAX TRACE ;
LINK class PT TRACE (IDENTIFICATEUR) ; integer IDENTIFICATEUR
   begin
        ref (HEAD) FILE TRACE ;
        character MARQUE ;
   FILE TRACE :- new HEAD ;
   end ;
LINK class TRANS (OPERATION, NOM, TEMPS TRACE) ;
     integer OPERATION ; text NOM ; real TEMPS TRACE
:
PROCESS class TRANSACTION ;
begin      :
         boolean TRACE ON ;
         integer ID ;
         character MARK TRACE ;
           :
procedure ENTER FACILITY(F) ; name F ; ref (FACILITY) F ;
    begin
      :
    if TRACE ON then SEARCH FILE TRACE (0,IDENT) ;
      :
    end ;
```

```
procedure SERACH FILE TRACE (OP, ID ENTITY) ; integer OP ;
                                              text ID ENTITY ;
 begin
    ref (PT TRACE) TT ;
 for TT :- TRANSACTIONQ.FIRST, TT.SUC while TT =/= none do
 inspect TT when PT TRACE do
 if IDENTIFICATEUR = ID then begin
      new TRANS (OP, ID ENTITY).INTO (FILE TRACE) ;
      MARQUE := MARK TRACE ;
                                end ;
 end ;

:
       - initialisation classe TRANSACTION -
     ID := ID + 1 ;
     if TRACE GEN then begin
          if  MAX TRACE >= ID then begin
                    TRACE ON := true ;
                    new PT TRACE (ID).INTO (TRANSACTIONQ) ;
                                        end ;
                    end ;
end classe TRANSACTION ;

:
procedure TRACE (N) ; integer N ;
 begin
    if not INIT TRACE then begin
              TRACE GEN := INIT TRACE := true ;
              MAX TRACE := N ;
                        end
                   else    - édition du TRACE - ;
 end ;

:
TRANSACTIONQ :- new HEAD ;
INNER
if  TRACE GEN then TRACE (0) ;
ERROR REPORT ;
end  GPSSS5 ;
```

CHAPITRE VI

CONCLUSIONS

Nous avons remarqué que GPSS était un langage hautement spécialisé et qu'un programmeur qui accepte le schéma de modélisation de GPSS peut spécifier un modèle avec un minimum d'efforts. L'envers de la médaille est qu'il est très lourd de spécifier des actions différentes de celles qui sont prévues dans le système.

Le problème est aggavé par l'âge et la nature spécialisée du langage. Les contraintes d'ecriture du texte du programme ave certaines données réservées pour les étiquettes, opérateurs et opérandes, rappelle les programmes d'assemblage. De plus, la notion de procédure avec paramètre est totalement inexistante et l'exécution de calculs demande des détours assez stupéfiants.
Ces quelques remarques soulignent que GPSS n'est pas un langage pour la programmation générale; par contre, il est excellent pour les applications auxquelles il est destiné.

A l'encontre de GPSS, SIMULA est un langage de programmation générale avec l'avantage majeur d'être extensible.
La classe standard SIMULATION donne à SIMULA, grâce aux concepts de temps simulé et de PROCESS analogue aux TRANSACTIONS de GPSS, le pouvoir de simuler des modèles bien plus vastes que ne pouvait résoudre le langage GPSS.

L'extension GPSSS5, rendant plus facile l'écriture du texte d'un programme de simulation, a repris les concepts de ressources et de file d'attente de GPSS qui étaient inexistants dans la classe SIMULATION.
De plus, par des procédures utilitaires, des statistiques détaillées analogues à celles de GPSS pouvait faciliter le travail du programmeur.

Un gros avantage, qui était absent à l'Université de Montréal, est que cette extension GPSSS5 soit sous forme précompilée sur le Dec 2050 de l'Université de Namur.

Faute de moyen, un prolongement possible du travail réalisé aurait pu être une analyse de performance d'exécution entre les langages GPSS et l'extention GPSSS5 de la classe SIMULATION de SIMULA.

°

° °

BIBLIOGRAPHIE

[1] P. BRATLEY et J. VAUCHER
"La programmation de simulation"
Ecole Internationale d'Eté d'Informatique de l'AFCET - Namur, juillet 1977.

[2] J. VAUCHER
"La programmation avec SIMULA 67"
Département d'Informatique. Facutlé des arts et des sciences.
Université de Montréal, juillet 1973.

[3] J. VAUCHER
"GPSS - User notes"
October 1974.

[4] *"SIMULA REFERENCE MANUAL"*, Control Data Corporation, publication 60234800.

[5] J. VAUCHER et D. DAVEY
"A wait until algorithm for general purpose simulation languages"
Département d'Informatique. Faculté des arts et des sciences, (juin 1972).

[6] J. VAUCHER
"Accélération du WAIT UNTIL pour l'ordonnancement conditionnel en simulation"
Département d'Informatique. Faculté des arts et des sciences.
Université de Montréal.

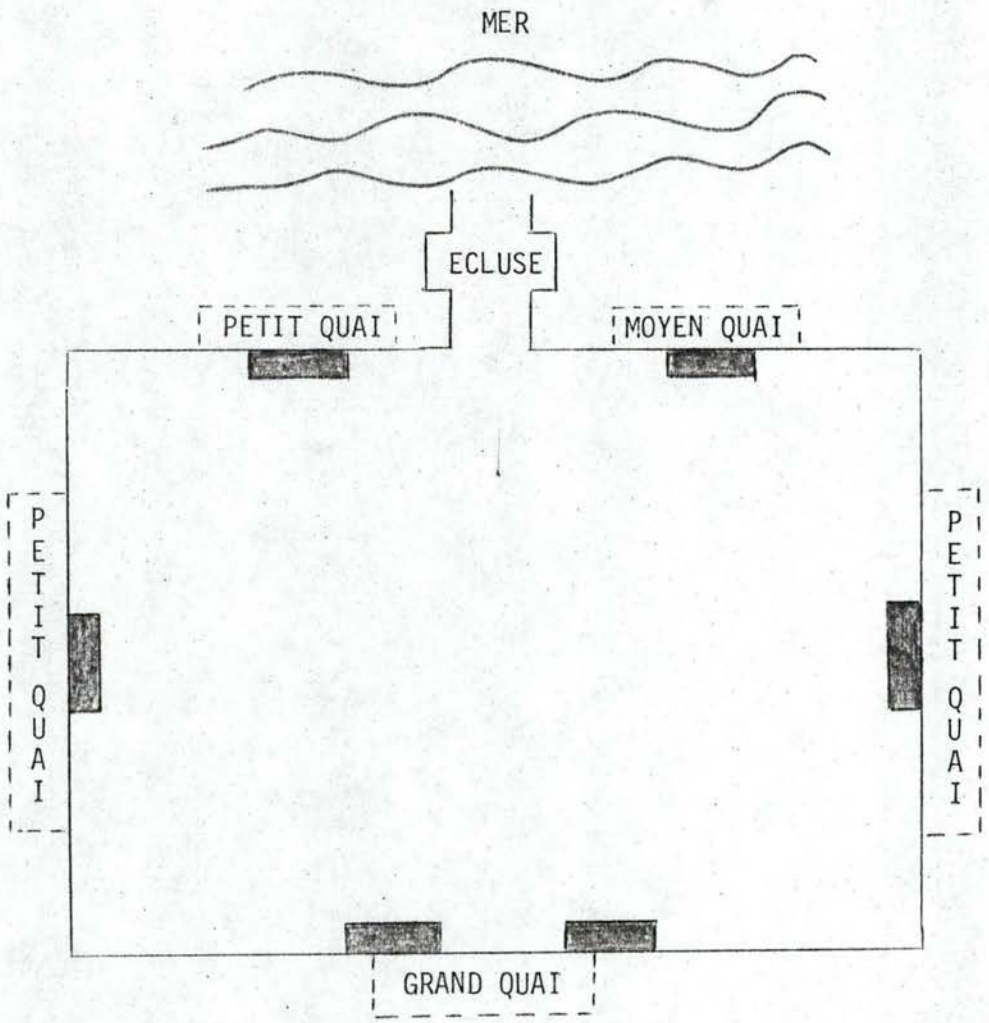
°

° °

ANNEXE A

EXEMPLE COMPLET D'UN PROGRAMME PRÉFIXÉ PAR G P S S S 5

EXEMPLE : SIMULATION D'UN PORT



: service de pompage

Quelques précisions au sujet des bateaux arrivant :

GROS BATEAUX

- Ils arrivent de façon aléatoire : 1 bateau/semaine
- Ils peuvent seulement utiliser le gros quai et nécessitent le service de deux pompes pour être déchargés, ce qui prend de 20 à 25 heures.

MOYEN BATEAUX

- Leur arrivée est plus fréquente, soit un bateau/3 jours.
- Ils peuvent utiliser un quai moyen ou un plus grand quai avec l'une de ces pompes. Le temps de déchargement est de 20 à 80 heures.

PETITS BATEAUX

- Il se présente en moyenne 1 bateau/jour.
- Ils peuvent utiliser n'importe quel quai n'utilisant alors qu'une seule pompe.

Le temps de déchargement est de 20 à 60 heures.

GESTION DU PORT

L'écluse peut servir un seul bateau à la fois, et il faut :

- 3 heures aux gros
- 2 heures aux moyens
- 1 heure aux petits

} bateaux pour franchir l'écluse.

Le grand quai peut supporter :

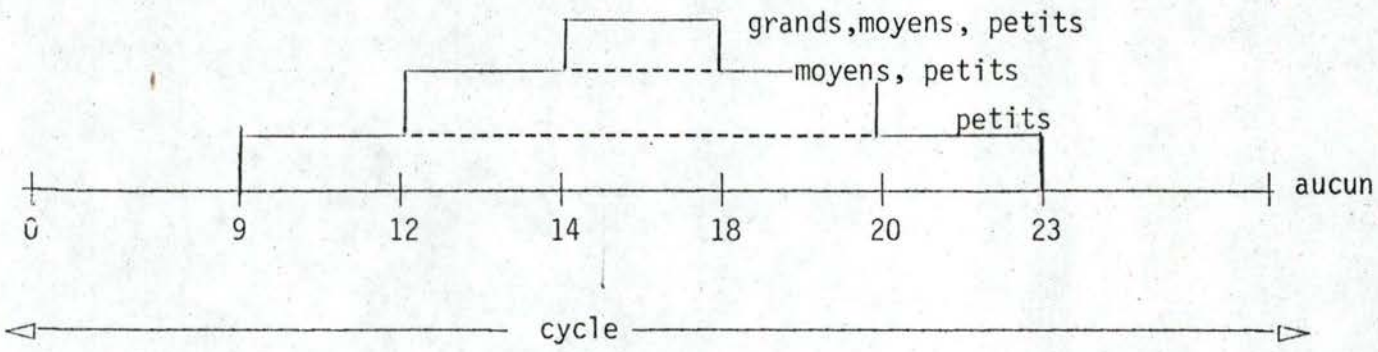
- un grand bateau
- 2 moyens bateaux
- 2 petits bateaux

Quant le bateau est dans le port, il faut 2 heures pour installer les pompes sur ce bateau.

Quand le bateau quitte le quai, il prend deux heures pour arriver à l'écluse.

CONTRAINTES DES MAREES

Les marées ont un cycle de 23 heures. Les bateaux qui peuvent alors utiliser l'écluse sont :



LA MARCHE GENERALE DE LA SIMULATION

1. Les bateaux arrivent.
2. Les bateaux entrent dans le port quand l'écluse est disponible et que la marée le permet.
3. Ils accostent le plus tôt possible avec les positions suivantes :
   - le gros bateau ira au grand quai
   - le bateau moyen ira vers le quai moyen si possible; si ce dernier est occupé, il ira vers le grand quai s'il n'y pas de grands bateaux en attente.
   - le petit bateau vers le petit quai si possible; sinon il ira vers le quai moyen; s'il n'y a pas de moyens bateaux en attente; sinon il ira vers le grand quai, s'il n'y a pas de grand bateaux en attente.
4. Quand le bateau est déchargé, il laisse le quai. Il quitte le port quand l'écluse est libre et que la marée est assez haute.

ETUDE PROPOSEE

- SIMULER pour une période de 2 semaines.
- Donner la trace des 10 premiers bateaux entrant dans le port.
- Intervalle de confiance.

o

o o

```
BEGIN
EXTERNAL  CLASS  GPSSS5  ;
GPSSS5   BEGIN

  REF (FACILITY) ECLUSE, MOYEQUAI  ;
  REF (STORAGE) GRANQUAI, PETIQUAI ;
  REF (REGION)   ECLUSER  ;
  REF (TABLE)    DISTGROS, DISTMOYEN, DISTPETIT  ;
  INTEGER  I, U1, U2, U3, U4, U5, U6  ;

TRANSACTION  CLASS  BATEAU (N) ;  INTEGER N  ;
BEGIN
     INTEGER  HEURE_ARRIVEE, HEURE_ATTENTE, MODULO  ;

COMMENT   ************************************ ;
          PROCEDURE  ENTREESORTIE  ;
COMMENT   ************************************ ;
BEGIN
     ENTER_REGION (ECLUSER)  ;
MAREE :   HEURE_ATTENTE := TIME ;
          MODULO := MOD (HEURE_ATTENTE,24) ;
          IF (MODULO>=14 AND MODULO<=18)
          OR (MODULO>=12 AND MODULO<=20 AND N<=2)
          OR (MODULO>= 9 AND MODULO<=23 AND N =1)
          THEN  GO  TO  ENTREE ;
          HOLD(1) ;
          GO  TO  MAREE  ;
ENTREE :  ENTER_FACILITY (ECLUSE) ;
          HOLD (N) ;
          LEAVE_FACILITY (ECLUSE) ;
     LEAVE_REGION (ECLUSER)  ;
  END  PROCEDURE ENTREESORTIE ;

COMMENT   ************************************ ;
          PROCEDURE GRANDQUAI (CAP,A)  ;
               INTEGER CAP, A ;
COMMENT   ************************************ ;
BEGIN
          ENTER_STORAGE (GRANQUAI,CAP) ;
          HOLD (2 + UNIFORM(20,A,U1))  ;
          LEAVE_STORAGE (GRANQUAI,CAP) ;
END PROCEDURE GRANDQUAI  ;

COMMENT   ************************************ ;
          PROCEDURE MOYENQUAI (B) ;
               INTEGER B ;
COMMENT   ************************************ ;
BEGIN
          ENTER_FACILITY (MOYEQUAI) ;
          HOLD (2 + UNIFORM(20,B,U2)) ;
          LEAVE_FACILITY (MOYEQUAI) ;
END PROCEDURE MOYENQUAI  ;

COMMENT   ************************************ ;
          PROCEDURE PETITQUAI (C) ;
               INTEGER C ;
COMMENT   ************************************ ;
BEGIN
          ENTER_STORAGE (PETIQUAI,1) ;
          HOLD (2 + UNIFORM(20,C,U3));
```

```
          LEAVE_STORAGE (PETIQUAI,1) ;
END PROCEDURE PETITQUAI ;

COMMENT   ************************************ ;
          PROCEDURE CHOIXQUAI (D) ;
                INTEGER D ;
COMMENT   ************************************ ;
BEGIN
          IF  CONTENTS_FACILITY (MOYEQUAI)  =  1
          THEN  BEGIN
                IF  CONTENTS_STORAGE (GRANQUAI) = 2
                THEN  MOYENQUAI (D)
                ELSE  GRANDQUAI (1,D) ;
                END
          ELSE  MOYENQUAI (D) ;
END PROCEDURE CHOIX ;

COMMENT  debut de la classe BATEAU ;

IF  N = 3  THEN  BEGIN
                 MARK_TRACE := 'G' ;
                 ACTIVATE NEW BATEAU (3) DELAY NEGEXP(1/168,U4) ;
                 END ;
IF  N = 2  THEN  BEGIN
                 MARK_TRACE := 'M' ;
                 ACTIVATE NEW BATEAU (2) DELAY NEGEXP(1/72,U5) ;
                 END
           ELSE  BEGIN
                 MARK_TRACE := 'P' ;
                 ACTIVATE NEW BATEAU (1) DELAY NEGEXP(1/24,U6) ;
                 END ;
HEURE_ARRIVEE := TIME ;
ENTREESORTIE  ;
IF  N = 3  THEN  GRANDQUAI(2,50) ;
IF  N = 2  THEN  CHOIXQUAI (80)
           ELSE  BEGIN
                 IF  CONTENTS_STORAGE (PETIQUAI) = 3
                 THEN  CHOIXQUAI (80)
                 ELSE  PETITQUAI (80) ;
                 END ;
COMMENT retour a l'ecluse ;
HOLD (2) ;

COMMENT sortie de l'ecluse ;
ENTREESORTIE ;

COMMENT prise de statistiques du temps ecoule entre l'entree et
        la sortie de l'ecluse ;
IF  N = 3  THEN  DISTGROS.ADD (TIME - HEURE_ARRIVEE) ;
IF  N = 2  THEN  DISTMOYEN.ADD(TIME - HEURE_ARRIVEE)
           ELSE  DISTPETIT.ADD(TIME - HEURE_ARRIVEE) ;

END  CLASS  BATEAU  ;

COMMENT  initialisation du programme utilisateur  ;

U1 := 55407 ;   U2 := 70455 ;   U3 := 757031 ;
U4 := 954072 ;  U5 := 270459 ;  U6 := 130757 ;
```

```
DISTGROS :- NEW TABLE ("GROS BATEAU",10,20,10) ;
DISTMOYEN:- NEW TABLE ("MOYEN BATEAU",10,20,10) ;
DISTPETIT:- NEW TABLE ("PETIT BATEAU",10,20,10) ;
ECLUSE   :- NEW FACILITY ("ECLUSE") ;
MOYEQUAI :- NEW FACILITY ("MOYEQUAI") ;
GRANQUAI :- NEW STORAGE  ("GRANQUAI",2) ;
PETIQUAI :- NEW STORAGE  ("PETIQUAI",3) ;
ECLUSER  :- NEW REGION   ("ECLUSER") ;

COMMENT  appel a la procedure INTERVALLE DE CONFIANCE ;
          INTER_CONFIANCE (ECLUSE,.2,4,0) ;

COMMENT   appel a la procedure TRACE ;
        TRACE (10) ;

COMMENT  appel a la procedure observation automatique ;
          OBSERVATION_AUTOMATIQUE (5) ;

COMMENT   activation des 3 premiers bateaux ;
        FOR I := 1 STEP 1 UNTIL 3 DO
         ACTIVATE NEW BATEAU (I) DELAY 0 ;

COMMENT   simuler pendant 2 semaines ;
        HOLD (24 * 14)  ;

COMMENT  demande du rapport de simulation ;
        STANDARD_REPORT  ;


END  GPSSS5 ;
END  BLOC EXTERIEUR
```

```
**********************
***  TRACE REPORT  ***
**********************


TRANSACTION      1  TYPE IS   P
-----------      -  ---- --   -
ENTERED   ECLUSE          AT TIME       9
LEAVED    ECLUSE          AT TIME      10
ENTERED   PETIQUAI        AT TIME      10
LEAVED    PETIQUAI        AT TIME      68
ENTERED   ECLUSE          AT TIME      70
LEAVED    ECLUSE          AT TIME      71

TRANSACTION      2  TYPE IS   M
-----------      -  ---- --   -
ENTERED   ECLUSE          AT TIME      12
LEAVED    ECLUSE          AT TIME      16
ENTERED   GRANQUAI        AT TIME      16
LEAVED    GRANQUAI        AT TIME      56
ENTERED   ECLUSE          AT TIME      60
LEAVED    ECLUSE          AT TIME      62

TRANSACTION      3  TYPE IS   P
-----------      -  ---- --   -
ENTERED   ECLUSE          AT TIME      14
LEAVED    ECLUSE          AT TIME      20
ENTERED   GRANQUAI        AT TIME      20
LEAVED    GRANQUAI        AT TIME     132
ENTERED   PETIQUAI        AT TIME     132
LEAVED    PETIQUAI        AT TIME     195
ENTERED   ECLUSE          AT TIME     204
LEAVED    ECLUSE          AT TIME     209

TRANSACTION      4  TYPE IS   M
-----------      -  ---- --   -
ENTERED   ECLUSE          AT TIME      12
LEAVED    ECLUSE          AT TIME      14
ENTERED   MOYENQUAI       AT TIME      14
LEAVED    MOYENQUAI       AT TIME      73
ENTERED   ECLUSE          AT TIME      84
LEAVED    ECLUSE          AT TIME      86

TRANSACTION      5  TYPE IS   P
-----------      -  ---- --   -
ENTERED   ECLUSE          AT TIME      13
LEAVED    ECLUSE          AT TIME      17
ENTERED   PETIQUAI        AT TIME      17
LEAVED    PETIQUAI        AT TIME      80
ENTERED   ECLUSE          AT TIME      82
LEAVED    ECLUSE          AT TIME      84

TRANSACTION      6  TYPE IS   P
-----------      -  ---- --   -
ENTERED   ECLUSE          AT TIME      15
LEAVED    ECLUSE          AT TIME      21
ENTERED   PETIQUAI        AT TIME      21
LEAVED    PETIQUAI        AT TIME      77
ENTERED   ECLUSE          AT TIME      81
LEAVED    ECLUSE          AT TIME      83
```

```
TRANSACTION       7  TYPE IS   P
-----------------------------

ENTERED    ECLUSE              AT TIME        38
LEAVED     ECLUSE              AT TIME        39
ENTERED    GRANQUAI            AT TIME        39
LEAVED     GRANQUAI            AT TIME        98
ENTERED    ECLUSE              AT TIME       105
LEAVED     ECLUSE              AT TIME       106

TRANSACTION       8  TYPE IS   P
-----------------------------

ENTERED    ECLUSE              AT TIME        69
LEAVED     ECLUSE              AT TIME        70
ENTERED    PETIQUAI            AT TIME        70
LEAVED     PETIQUAI            AT TIME       124
ENTERED    ECLUSE              AT TIME       129
LEAVED     ECLUSE              AT TIME       130

TRANSACTION       9  TYPE IS   P
-----------------------------

ENTERED    ECLUSE              AT TIME        81
LEAVED     ECLUSE              AT TIME        83




ENTERED    PETIQUAI            AT TIME        93
LEAVED     PETIQUAI            AT TIME       100
ENTERED    ECLUSE              AT TIME       110
LEAVED     ECLUSE              AT TIME       111

TRANSACTION      10  TYPE IS   P
-----------------------------

ENTERED    ECLUSE              AT TIME        90
LEAVED     ECLUSE              AT TIME        91
ENTERED    PETIQUAI            AT TIME        91
LEAVED     PETIQUAI            AT TIME       163
ENTERED    ECLUSE              AT TIME       165
LEAVED     ECLUSE              AT TIME       166

3 Garbage collection(s) in 151 ms


MAX.TRACE   10

End of SIMULA program execution.
CPU time: 0.53  Elapsed time: 2:25.00
@
```

```
*************************************
***   INTERVALLE DE CONFIANCE     ***
*************************************
```

TYPE D'OBSERVATIONS : NOMBRE MOYEN DE TRANSACTIONS EN ATTENTE
PARAMETRE DEMANDE DE COVARIANCE : 0.200
LONGUEUR DEMANDEE DE L'INTERVALLE : 4.000

NB D'OB.	NB BLOCS	METHODE	LONGUEUR BIENAYME	LONGUEUR NORMALE
4		1	2.903	
5		1	3.399	
6		1	2.040	
7		1	1.774	
8	0	2	4.741	

ECART TYPE : 1.041
LONGUEUR DEMANDEE : 4.000
LONGUEUR REELLE : 4.741
AL

```
*****************************   VERSION 5.0
***   MONTREAL  GPSS     ***
***  SIMULATION REPORT   ***
*****************************

PASSE =  1

START TIME=      0.00
END   TIME=     35.00


* FACILITIES *
*************
```

	AVERAGE UTILISATION	ENTRIES	AVERAGE TRANSIT TIME	STATUS
ECLUSE	0.29	6	1.67	FREE
MOYENQUAI	0.60	1	21.06	BUSY

FILE D'ATTENTE : FACILITIES

	NOMBRE D'OBS.	NOMBRE MOYEN	INTERVALLE MIN	INTERVALLE MAX	TEMPS MOYEN	INTERVALLE MIN	INTERVALLE MAX
ECLUSE	8	0.375	4.364	5.116	2.203	1.217	5.623
MOYCQUAI	7	0.000	0.000	0.000	0.000	0.000	0.000

* STORAGES *

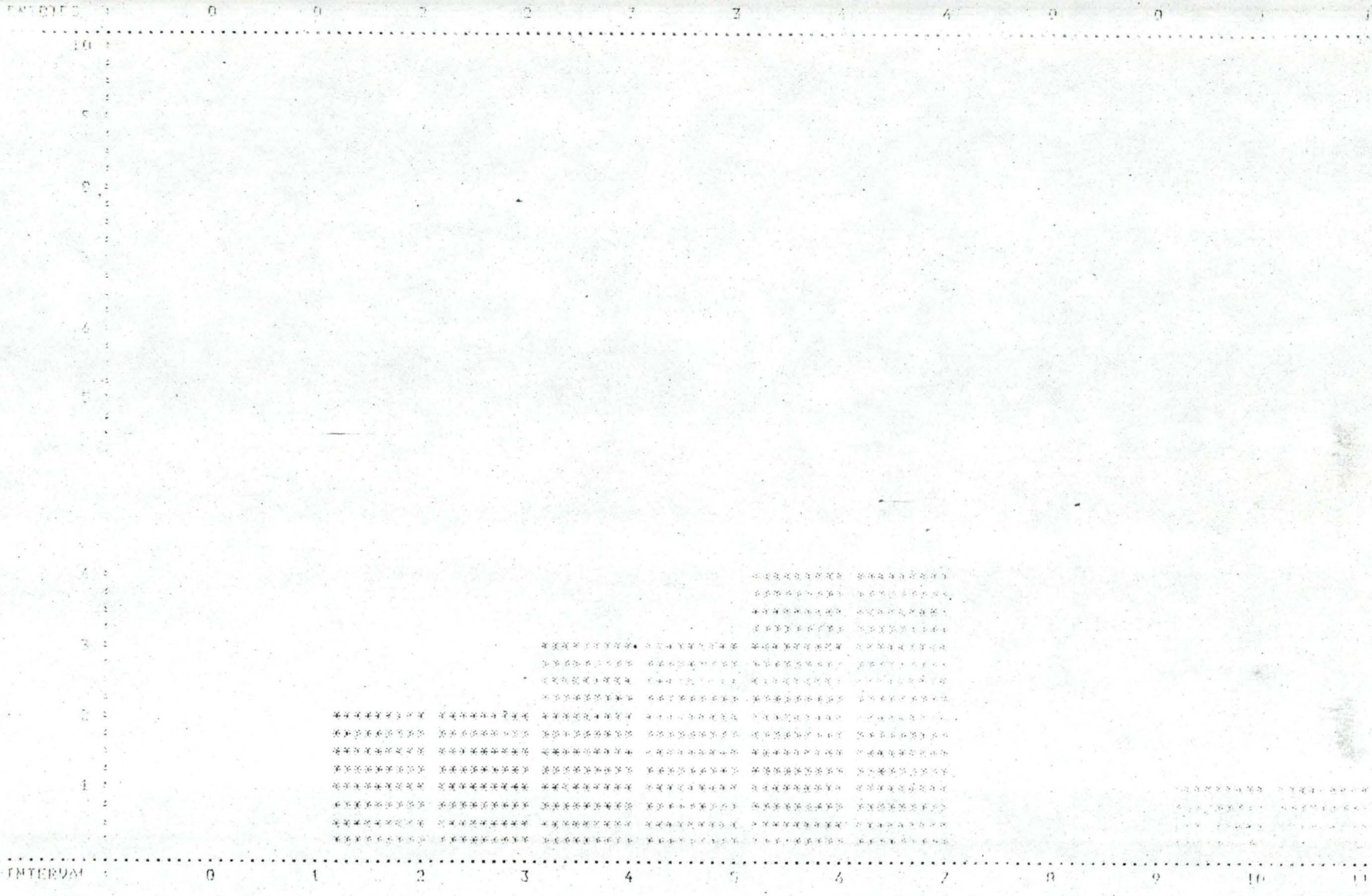
	CAPACITY	AVG. CONTENTS	AVG. UTIL.	ENTRIES	THRU TIME / UNIT	CONTENTS MAX	CONTENTS NOW
FRANQUAI	2	0.54	0.2723	4	19.06	1	1
PETIQUAI	3	1.63	0.5440	3	19.04	3	3

FILE D'ATTENTE : STORAGES

	NOMBRE D'OBS.	NOMBRE MOYEN	INTERVALLE MIN	INTERVALLE MAX	TEMPS MOYEN	INTERVALLE MIN	INTERVALLE MAX
FRANQUAI	7	0.429	0.409	1.265	0.000	0.000	0.000
PETIQUAI	7	0.000	0.000	0.000	0.000	0.000	0.000

* REGIONS *

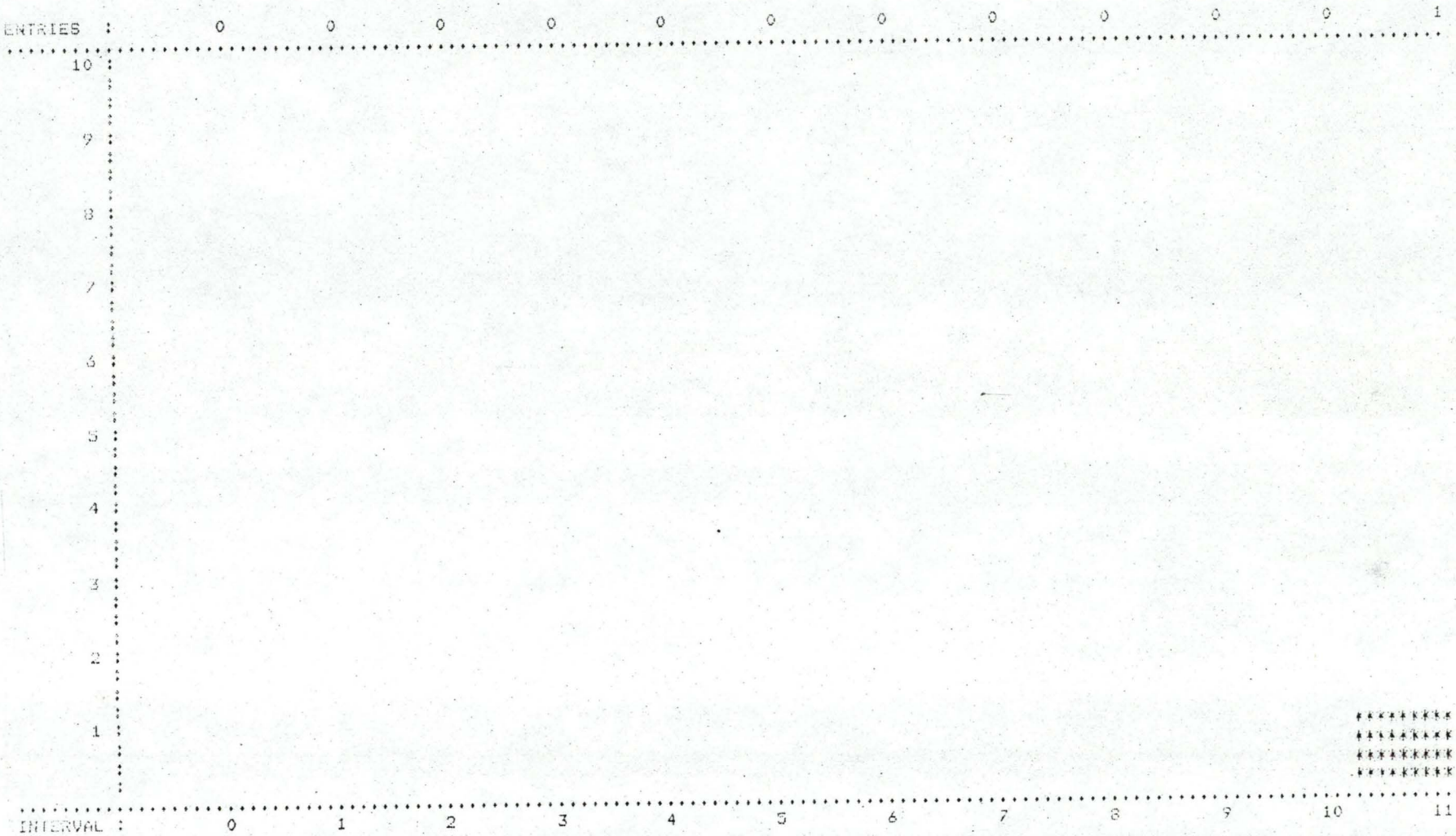
	ENTRIES	ZERO_ENTRIES	MAX. CONTENTS	CURRENT CONTENTS	AVG. CONTENTS	AVG. TIME/TRANS ALL	AVG. TIME/TRANS NON ZERO
ECLUSER	4	0	4	0	1.66	9.70	9.70



INTERVAL WIDTH = 10.000
INTERVAL NO. 1 = 20.000 TO 30.000
INTERVAL NO. 10 = 110.000 TO 120.000

TOTAL ENTRIES = 20
AVERAGE = 73.320
MIN VALUE = 32.368
MAX VALUE = 208.712

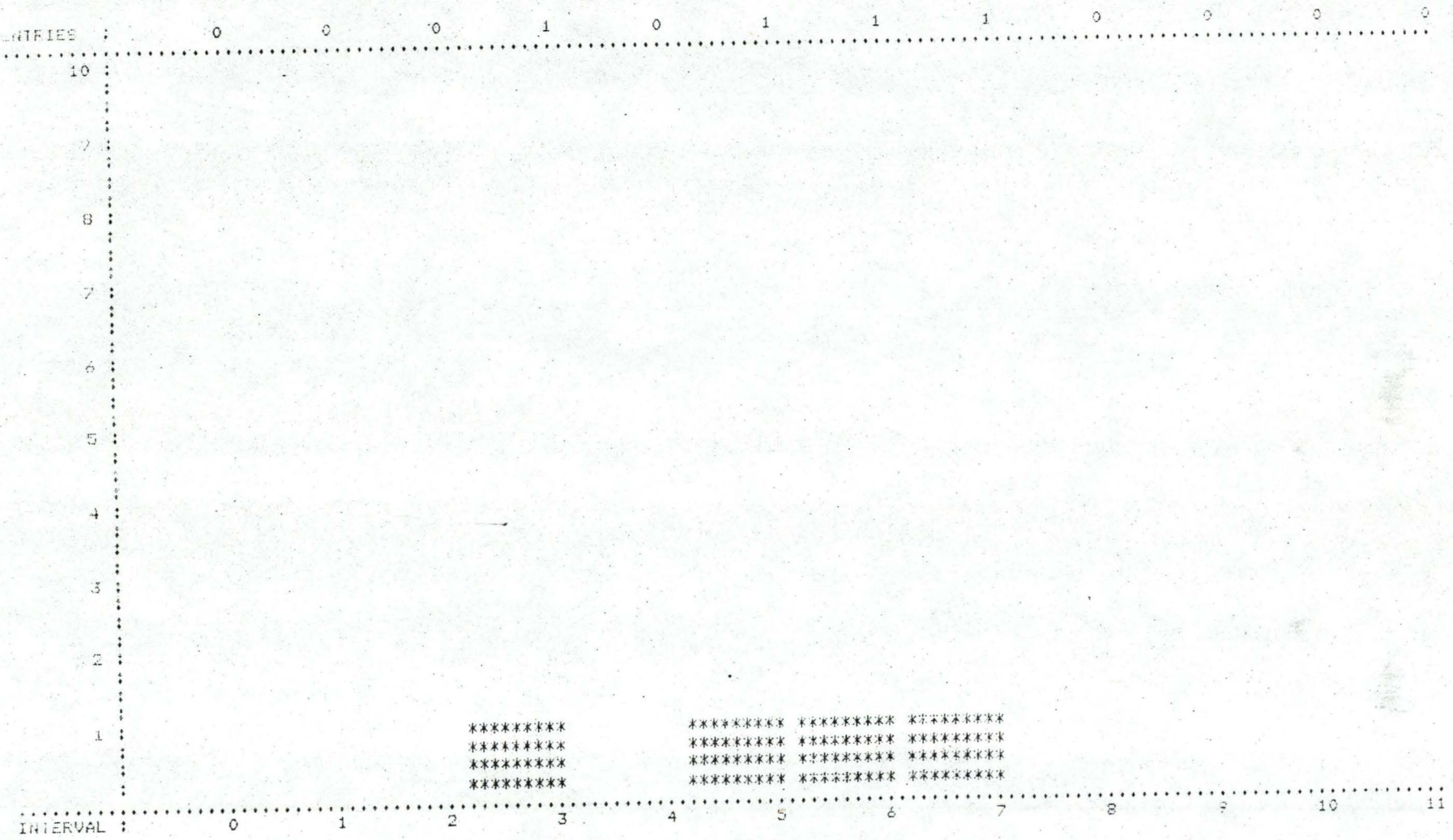
* TABLE * GROS BATEAU
****



INTERVAL WIDTH = 10.000
INTERVAL NO. 1 = 20.000 TO 30.000
INTERVAL NO. 10 = 110.000 TO 120.000

TOTAL ENTRIES = 1
AVERAGE = 208.712
MIN VALUE = 208.712
MAX VALUE = 208.712

* TABLE * MOYEN BATEAU
****



INTERVAL WIDTH = 10.000
INTERVAL NO. 1 = 20.000 TO NTERVAL NO. 110.000 12

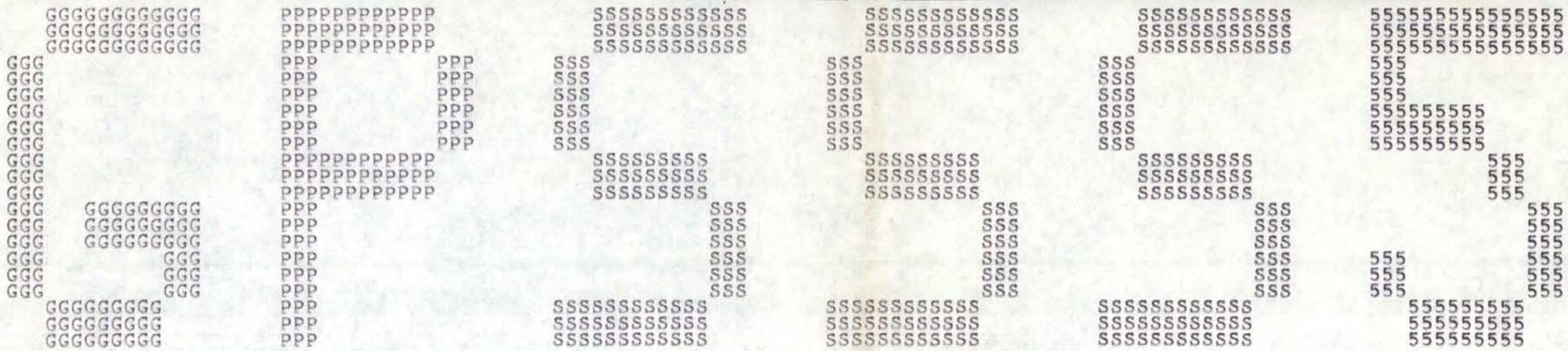
TOTAL ENTRIES = 4

AVERA VA= 46.809
MAX =

ANNEXE B

LISTING DE L'EXTENSION G P S S S 5

ANNEXE B. Listing de l'extension de GPSSS5 : listing joint en annexe.



FM B 16/1979/8

VAN IMPE

START User INF.M-VANIMPE Job GPSSS5 Seq. 5582 Date 21-Sep-79 12:39:09 Monitor SOPHIE * FUNDP Namur Belgique * DEC *START*
File: PS:<INF.M-VANIMPE>GPSSS5.SIM.258 Created: 20-Sep-79 15:27:04 Printed: 21-Sep-79 12:44:28
QUEUE Switches: /COPIES:5 /SPACING:0 /LIMIT:480 /FORMS:NORMAL

18/2/69

```
00050    COMMENT ************************************************
00100
00150            MONTREAL  GPSSS PACKAGE + VERSION 5.0 +
00200
00250          WRITTEN - FEBRUARY 1972   VERSION 4.0
00300          UPDATED - SEPTEMBER 1974  VERSION 4.1
00350          UPDATED - MAY 1977        VERSION 5.0
00400          AUTHOR  - JEAN G. VAUCHER
00450                    PROFESSEUR AGREGE
00500                    DEPARTEMENT D INFORMATIQUE
00550                    UNIVERSITE DE MONTREAL
00600                    C.P. 6128, SUCCURSALE A
00650                    MONTREAL, CANADA
00700                    H3C 3J7
00750
00800    ************************************************************;
00850
00900     SIMULATION CLASS  GPSSS5 ;
00950     BEGIN
01000
01050        REF  (HEAD) FACILITYQ,
01100                    REGIONQ,
01150                    STORAGEQ,
01200                    TRANSACTIONQ,
01250                    TABLEQ ;
01300
01350        INTEGER  TRANS_ID, PASSE, U ;
01400        REAL  TIME_ORIGIN, SIMULATION_START_TIME ;
01450
01500      LINK CLASS  TRANS (OPERATION,NOM,TEMPS_TRACE) ;  VALUE  OPERATION  ;
01550       VALUE NOM; VALUE TEMPS_TRACE; INTEGER OPERATION;
01600       TEXT NOM ;  REAL TEMPS_TRACE ; ;
01650
01700      LINK  CLASS PT_TRACE (IDENTIFICATEUR) ; INTEGER IDENTIFICATEUR ;
01750         BEGIN
01800                    REF (HEAD)  FILE_TRACE ;
01850                    CHARACTER   MARQUE     ;
01900                    BOOLEAN     FILE_CREATED  ;
01950                    FILE_TRACE :-  NEW  HEAD
02000         END  ;
02050
02100      LINK  CLASS OBS (X) ; VALUE X ; REAL X  ; ;
02150
02200      LINK  CLASS  ENTITY (IDENT) ;  VALUE  IDENT ;  TEXT  IDENT ;
02250     IF IDENT.LENGTH > 15 THEN  IDENT :- IDENT.SUB(1,15) ;
02300
02350
02400
02450
02500
02550     COMMENT ************************************************
02600
02650              FACILITY DEFINITION
02700
02750    ************************************************************;
02800
02850    ENTITY  CLASS  FACILITY ;
02900     BEGIN
02950        REF (HEAD) INQ ;
03000        REF (TRANSACTION) OCCUPIER ;
```

```
03050       REF (HEAD) LISTE, LISTE2 ;
03100       INTEGER ENTRIES  ;
03150       REAL  TLAST, BUSY_TIME ;
03200      LISTE :- NEW HEAD ;
03250      LISTE2 :- NEW HEAD ;
03300      INQ :-  NEW  HEAD ;
03350      TLAST := TIME_ORIGIN ;
03400      INTO (FACILITYQ)
03450
03500    END  FACILITY DEFINITION ;
03550
03600    COMMENT ****************************************************
03650
03700            STORAGE DEFINITION
03750
03800    ************************************************************;
03850
03900    ENTITY  CLASS  STORAGE (CAPACITY) ;  INTEGER  CAPACITY ;
03950     BEGIN
04000        REF  (HEAD) INQ ;
04050        REF  (HEAD) LISTE, LISTE2 ;
04100        INTEGER MAX, ENTRIES, UNIT_ENTRIES ;
04150        REAL  CONTENTS, INTGRL, TLAST ;
04200
04250        PROCEDURE  CHECK_INQ ;
04300        BEGIN
04350           REF  (TRANSACTION) CLIENT, NEXT_CLIENT ;
04400           CLIENT :- INQ.FIRST ;
04450           WHILE  CLIENT =/=  NONE   AND  CONTENTS  \=  CAPACITY  DO
04500              BEGIN
04550                NEXT_CLIENT :- CLIENT.SUC ;
04600                 ACTIVATE  CLIENT ;
04650                CLIENT :- NEXT_CLIENT
04700              END
04750        END  CHECK_INQ ;
04800
04850      INQ :-  NEW  HEAD ;
04900      LISTE :-  NEW  HEAD ;
04950      LISTE2 :- NEW HEAD  ;
05000      TLAST := TIME_ORIGIN ;
05050      INTO (STORAGEQ)
05100     END  STORAGE DEFINITION ;
05150
05200     COMMENT ****************************************************
05250
05300            REGION DEFINITION
05350
05400    ************************************************************;
05450
05500    ENTITY  CLASS  REGION ;
05550     BEGIN
05600        INTEGER  ENTRIES, ZERO_ENTRIES, MAX ;
05650        REAL  CONTENTS, INTGRL, TLAST ;
05700
05750      TLAST := TIME_ORIGIN ;
05800      INTO (REGIONQ)
05850     END  REGION DEFINITION ;
05900
05950     COMMENT ****************************************************
06000
```

```
06050             GROUP DEFINITION
06100
06150    ************************************************************;
06200
06250     CLASS  GROUP (TAILLE) ;  INTEGER  TAILLE ;
06300     BEGIN
06350        INTEGER  N ;
06400        REF  (HEAD) INQ ;
06450
06500        BOOLEAN   PROCEDURE  READY ;
06550       READY := (N+1 = TAILLE) ;
06600
06650       INQ :-  NEW  HEAD
06700     END  GROUP DEFINITION ;
06750
06800     COMMENT ***************************************************
06850
06900             TABLE DEFINITION
06950
07000    ************************************************************;
07050
07100    ENTITY  CLASS  TABLE (N, LOWER, INTER) ;
07150     INTEGER  N ;
07200     REAL  LOWER, INTER ;
07250     BEGIN
07300        INTEGER   ARRAY  A [-1 : N] ;
07350        INTEGER  ENTRIES ;
07400        REAL  MIN, MAX, SUM ;
07450
07500        PROCEDURE  ADD (X) ;  REAL  X ;
07550        BEGIN
07600           INTEGER  INDEX ;
07650          SUM := SUM + X ;
07700          ENTRIES := ENTRIES + 1 ;
07750           IF  ENTRIES = 1  THEN  MIN := MAX := X
07800           ELSE   IF  X < MIN  THEN  MIN := X
07850           ELSE   IF  X > MAX  THEN  MAX := X ;
07900          INDEX := ENTIER((X - LOWER) / INTER) ;
07950           IF  INDEX < 0  THEN  A[-1] := A[-1] + 1
08000           ELSE   IF  INDEX > N  THEN  A[N] := A[N] + 1
08050           ELSE  A[INDEX] := A[INDEX] + 1
08100        END  ;
08150
08200       INTO (TABLEQ)
08250     END  TABLE DEFINITION ;
08300
08350     COMMENT ***************************************************
08400
08450             TRANSACTION DEFINITION
08500
08550    ************************************************************;
08600
08650    PROCESS  CLASS  TRANSACTION ;
08700     BEGIN
08750        BOOLEAN  TRACE_ON ;
08800        INTEGER  ID ;
08850        REAL  TIME_MARK, PRIORITY, LAST_REGION_ENTRY_TIME ;
08900        REAL  TEMPS_ENTREE ;
08950        CHARACTER MARK_TRACE ;
09000        REF  (REGION) LAST_REGION ;
```

```
09050
09100   COMMENT          :::::::::::::::::::::::::::::::::::::::
09150                    ::: PROCEDURE SEARCH_FILE_TRACE     :::
09200                    :::::::::::::::::::::::::::::::::::::::;
09250
09300   PROCEDURE SEARCH_FILE_TRACE (OP,ID_ENTITY)     ; VALUE ID_ENTITY ;
09350             INTEGER OP ;  TEXT  ID_ENTITY  ;
09400   BEGIN
09450       REF (PT_TRACE) TT ;
09500       FOR TT :- TRANSACTIONQ.FIRST, TT.SUC  WHILE TT =/= NONE DO
09550       INSPECT  TT  WHEN  PT_TRACE  DO
09600         IF IDENTIFICATEUR = ID THEN BEGIN
09650                NEW TRANS (OP,ID_ENTITY,TIME).INTO(FILE_TRACE) ;
09700                FILE_CREATED := TRUE  ;
09750                MARQUE :=  MARK_TRACE ;
09800                GO TO FIN
09850                                    END ;
09900   FIN :  END ;
09950
10000
10050
10100    COMMENT ************
10150
10200        ENTER_FACILITY
10250
10300   ********************;
10350
10400       PROCEDURE  ENTER_FACILITY (F) ;  NAME  F ;  REF  (FACILITY) F ;
10450       BEGIN
10500          IF  F ==  NONE   THEN  F :-  NEW  FACILITY ("ERR") ;
10550          INSPECT  F  DO
10600             BEGIN
10650                IF TRACE_ON  THEN  SEARCH_FILE_TRACE (0,IDENT) ;
10700                IF NOT COLLECT_AUTOM  THEN  BEGIN
10750                    NEW OBS (WAITING_FACILITY(F)). INTO (LISTE);
10800                    IF I_C_NOM ==  F     AND  I_C_TYP = 0
10850                                         AND  ARRET_SIMULATION
10900                    THEN ACTIVATE NEW INTERVALLE (LISTE,LISTE.CARDINAL)
10950                                              END ;
11000               TEMPS_ENTREE := TIME ;
11050                IF  OCCUPIER ==  THIS  TRANSACTION  THEN  ERREUR (6,F)
11100                ELSE   IF  OCCUPIER =/=  NONE   THEN
11150                   BEGIN
11200                     PRIORITY_INTO (INQ) ;
11250                     PASSIVATE ;
11300                      THIS  TRANSACTION . OUT
11350                   END
11400                ELSE
11450                   BEGIN
11500                     OCCUPIER :-  THIS  TRANSACTION ;
11550                     TLAST := TIME
11600                   END  ;
11650                ENTRIES := ENTRIES + 1 ;
11700                NEW OBS (TIME - TEMPS_ENTREE).INTO (LISTE2) ;
11750                IF I_C_NOM ==   F     AND  I_C_TYP = 1
11800                                      AND  ARRET_SIMULATION
11850                THEN ACTIVATE NEW INTERVALLE (LISTE2,LISTE2.CARDINAL) ;
11900             END
11950       END   ENTER_FACILITY ;
12000
```

```
12050    COMMENT ************
12100
12150         LEAVE_FACILITY
12200
12250    ********************;
12300
12350        PROCEDURE  LEAVE_FACILITY (F) ;  REF  (FACILITY) F ;
12400        BEGIN
12450           INSPECT  F  DO
12500              BEGIN
12550               IF  TRACE_ON  THEN  SEARCH_FILE_TRACE (1,IDENT) ;
12600               IF  NOT COLLECT_AUTOM  THEN  BEGIN
12650                    NEW OBS (WAITING_FACILITY(F)), INTO (LISTE) ;
12700                    IF I_C_NOM ==   F     AND  I_C_TYP = 0
12750                                          AND  ARRET_SIMULATION
12800                    THEN ACTIVATE NEW INTERVALLE (LISTE,LISTE.CARDINAL) ;
12850                                               END ;
12900              IF  OCCUPIER =/=  THIS  TRANSACTION  THEN  ERREUR (3,F)
12950              ELSE
13000                 BEGIN
13050                   OCCUPIER :- INQ.FIRST ;
13100                   IF  OCCUPIER ==  NONE   THEN
13150                      BUSY_TIME := BUSY_TIME + TIME - TLAST
13200                   ELSE   ACTIVATE  OCCUPIER  DELAY  0
13250                 END
13300              END
13350
13400           OTHERWISE  ERREUR (10, NONE )
13450        END   LEAVE_FACILITY ;
13500
13550    COMMENT ************
13600
13650         ENTER_STORAGE
13700
13750    ********************;
13800
13850        PROCEDURE  ENTER_STORAGE (S, REQUIRED) ;
13900        NAME  S ;  REF  (STORAGE) S ;
13950        INTEGER  REQUIRED ;
14000        BEGIN
14050           IF  S ==  NONE   THEN  S :-  NEW  STORAGE ("ERR", 1000000) ;
14100           INSPECT  S  DO
14150            BEGIN
14200               IF TRACE_ON  THEN SEARCH_FILE_TRACE (0,IDENT) ;
14250               IF NOT COLLECT_AUTOM  THEN  BEGIN
14300                   NEW OBS (WAITING_STORAGE(S)), INTO (LISTE) ;
14350                   IF I_C_NOM ==   S     AND  I_C_TYP = 0
14400                                         AND  ARRET_SIMULATION
14450                   THEN ACTIVATE NEW INTERVALLE (LISTE,LISTE.CARDINAL) ;
14500                                              END ;
14550               IF  REQUIRED > CAPACITY  THEN  ERREUR (4,S)
14600               ELSE
14650                  BEGIN
14700                    PRIORITY_INTO (INQ) ;
14750                    TEMPS_ENTREE := TIME ;
14800                    WHILE  CONTENTS + REQUIRED > CAPACITY  DO  PASSIVATE ;
14850                    NEW OBS (TIME - TEMPS_ENTREE),INTO (LISTE2) ;
14900                    IF I_C_NOM ==  S     AND  I_C_TYP = 1
14950                                         AND  ARRET_SIMULATION
15000                    THEN ACTIVATE NEW INTERVALLE (LISTE2,LISTE2.CARDINAL) ;
```

```
15050                 THIS  TRANSACTION . OUT ;
15100                 ENTRIES := ENTRIES + 1 ;
15150                 UNIT_ENTRIES := UNIT_ENTRIES + REQUIRED ;
15200                 ACCUM (INTGRL, TLAST, CONTENTS, REQUIRED) ;
15250                  IF  CONTENTS > MAX  THEN  MAX := CONTENTS ;
15300                 HOLD (0)
15350               END
15400           END
15450       END  ENTER_STORAGE ;
15500
15550    COMMENT ************
15600
15650        LEAVE_STORAGE
15700
15750    ********************;
15800
15850       PROCEDURE  LEAVE_STORAGE (S, RELEASED) ;
15900       REF  (STORAGE) S ;
15950       INTEGER  RELEASED ;
16000       INSPECT  S  DO
16050          BEGIN
16100            IF TRACE_ON THEN  SEARCH_FILE_TRACE (1,IDENT) ;
16150            IF NOT COLLECT_AUTOM  THEN  BEGIN
16200               NEW OBS (WAITING_STORAGE(S)), INTO (LISTE) ;
16250               IF I_C_NOM ==   S    AND  I_C_TYP = 0
16300                                    AND  ARRET_SIMULATION
16350               THEN ACTIVATE NEW INTERVALLE (LISTE,LISTE.CARDINAL) ;
16400                                    END ;
16450            ACCUM (INTGRL, TLAST, CONTENTS, -RELEASED) ;
16500             IF  CONTENTS < 0  THEN
16550               BEGIN
16600                 CONTENTS := 0 ;
16650                 ERREUR (5,S)
16700               END  ;
16750            CHECK_INQ
16800          END
16850       OTHERWISE  ERREUR (11, NONE ) ;
16900
16950    COMMENT ************
17000
17050        ENTER_REGION
17100
17150    ********************;
17200
17250       PROCEDURE  ENTER_REGION (R) ;  NAME  R ;  REF  (REGION) R ;
17300       BEGIN
17350          IF  R ==  NONE   THEN  R :-  NEW  REGION ("ERR") ;
17400         LAST_REGION :- R ;
17450         LAST_REGION_ENTRY_TIME := TIME ;
17500          INSPECT  R  DO
17550            BEGIN
17600              ENTRIES := ENTRIES + 1 ;
17650              ACCUM (INTGRL, TLAST, CONTENTS, 1) ;
17700               IF  CONTENTS > MAX  THEN  MAX := CONTENTS
17750            END
17800       END  ENTER_REGION ;
17850
17900    COMMENT ************
17950
18000        LEAVE_REGION
```

```
18050
18100   ********************;
18150
18200       PROCEDURE  LEAVE_REGION (R) ;  REF  (REGION) R ;
18250       INSPECT  R  DO
18300          IF  CONTENTS = 0  THEN  ERREUR (7,R)
18350          ELSE
18400             BEGIN
18450                IF  LAST_REGION == R  AND  LAST_REGION_ENTRY_TIME = TIME
18500                   THEN  ZERO_ENTRIES := ZERO_ENTRIES + 1 ;
18550              ACCUM (INTGRL, TLAST, CONTENTS, -1)
18600             END
18650       OTHERWISE  ERREUR (12, NONE ) ;
18700
18750    COMMENT ************
18800
18850        PRIORITY_INTO
18900
18950   ********************;
19000
19050       PROCEDURE  PRIORITY_INTO (Q) ;  REF  (HEAD) Q ;
19100       BEGIN
19150          REF  (TRANSACTION) P ;
19200         P :- Q.FIRST ;
19250          IF  P ==  NONE   THEN  INTO (Q)
19300          ELSE   IF  PRIORITY < P.PRIORITY  OR
19350                     (PRIORITY = P.PRIORITY  AND  PRIORITY < 0)
19400                  THEN  PRECEDE (P)
19450          ELSE
19500             BEGIN
19550               P :- Q.LAST ;
19600               WHILE PRIORITY < P.PRIORITY DO P :- P.PRED ;
19650               FOLLOW (P)
19700             END
19750       END   PRIORITY_INTO ;
19800
19850    COMMENT *********
19900
19950       WAIT_UNTIL
20000
20050   *****************;
20100
20150       PROCEDURE  WAIT_UNTIL (B) ;  NAME  B ;  BOOLEAN  B ;
20200       IF   NOT B  THEN
20250          BEGIN
20300               ACTIVATE  WAIT_MONITOR  AFTER  NEXTEV ;
20350            PRIORITY_INTO (WAITQ) ;
20400            PASSIVATE ;
20450             WHILE   NOT B  DO
20500                IF  SUC ==  NONE
20550                THEN  RESUME(WAIT_MONITOR)
20600                ELSE  RESUME(SUC);
20650            OUT ;
20700            WAIT_ACTION :=  TRUE ;
20750             ACTIVATE   THIS  PROCESS;
20800          END  WAIT_UNTIL ;
20850
20900    COMMENT ************
20950
21000          JOIN
```

```
21050
21100   ********************;
21150
21200       PROCEDURE  JOIN (G) ;  REF  (GROUP) G ;
21250       BEGIN
21300          REF  (PROCESS) P ;
21350          INSPECT  G  DO
21400             BEGIN
21450               N := N + 1 ;
21500               IF   N < TAILLE  THEN
21550                  BEGIN
21600                    WAIT(INQ) ;
21650                    OUT
21700                  END
21750               ELSE
21800                  BEGIN
21850                    N := 0 ;
21900                    P :- INQ.FIRST ;
21950                     WHILE  P =/=  NONE   DO
22000                        BEGIN
22050                          ACTIVATE  P  DELAY  0 ;
22100                          P :- P.SUC
22150                        END  ;
22200                    HOLD (0)
22250                  END
22300             END
22350          OTHERWISE  ERREUR (8, NONE )
22400       END  JOIN ;
22450
22500
22550      TIME_MARK := GPSSS_TIME ;
22600      ID := TRANS_ID := TRANS_ID + 1 ;
22650      IF TRACE_GEN  THEN  BEGIN
22700         IF  MAX_TRACE  >= ID
22750         THEN   BEGIN   TRACE_ON := TRUE  ;
22800                        NEW PT_TRACE (ID) .INTO (TRANSACTIONQ) ;
22850                 END ;
22900                        END ;
22950
23000    END  CLASS TRANSACTION ;
23050
23100
23150   COMMENT ****************************************
23200
23250              OBSERVATION  DEFINITION
23300
23350           ****************************************;
23400
23450    INTEGER  TIME_BTW_OBS  ;
23500
23550   PROCESS  CLASS  OBSERVATION  ;
23600    BEGIN
23650        REF (ENTITY) P ;
23700        ACTIVATE  NEW  OBSERVATION  DELAY  TIME_BTW_OBS ;
23750
23800    FOR P :- FACILITYQ.FIRST, STORAGEQ.FIRST  DO
23850    WHILE  P =/= NONE  DO  BEGIN
23900    INSPECT  P  WHEN  FACILITY  DO  BEGIN
23950                        NEW OBS (WAITING_FACILITY(P)).INTO (LISTE) ;
24000                        IF I_C_NOM ==   P    AND  I_C_TYP = 0
```

```
24050                                   AND  ARRET_SIMULATION
24100                          THEN ACTIVATE NEW INTERVALLE (LISTE,LISTE.CARDINAL)
24150                               END
24200                 WHEN  STORAGE   DO  BEGIN
24250                       NEW OBS (WAITING_STORAGE(P)) .INTO (LISTE) ;
24300                       IF I_C_NOM ==   P     AND  I_C_TYP = 0
24350                                         AND  ARRET_SIMULATION
24400                        THEN ACTIVATE NEW INTERVALLE (LISTE,LISTE.CARDINAL)
24450                               END;
24500     P :- P.SUC ;
24550                          END ;
24600
24650     END  CLASS  OBSERVATION  ;
24700
24750
24800
24850     COMMENT  ************************************************
24900
24950                 INTERVALLE  DEFINITION
25000
25050               ************************************************ ;
25100
25150       REAL  MOY , INTERMIN , INTERMAX  ;
25200
25250     PROCESS  CLASS   INTERVALLE (FILE,NB) ;       REF (HEAD) FILE ;
25300                                                   INTEGER    NB   ;
25350     BEGIN
25400           REF (OBS) Z ;
25450           INTEGER   I , J , K , NB_BLOCS , METHODE ;
25500           REAL ARRAY TAB[1:NB] ;
25550           REAL      NUM , DENOM , SOMME , VARIANCE , ECART_TYPE ;
25600
25650     COMMENT   remise a zero des variables globales  ;
25700           MOY := INTERMIN := INTERMAX := 0  ;
25750
25800     COMMENT   mise en table des observations et calcul de leur moyenne ;
25850           IF NB = 0  THEN  BEGIN
25900                              COMMENT OUTTEXT("NO OBSERVATIONS") ;
25950                              GO TO  FIN
26000                          END  ;
26050           FOR Z :- FILE.FIRST, Z.SUC WHILE Z=/=NONE DO
26100                          BEGIN
26150                              SOMME := SOMME + Z.X ;
26200                              I := I + 1  ;
26250                              TAB[I] := Z.X ;
26300                          END ;
26350           MOY  :=  SOMME / NB  ;
26400           SOMME  :=  0         ;
26450
26500     COMMENT   calcul du DENOM = 1/nb somme de I jusque NB (Xi-moy)**2 ;
26550           FOR I:=1 STEP 1 UNTIL NB  DO
26600                DENOM := DENOM + (TAB[I]-MOY)**2 ;
26650           IF DENOM = 0  THEN  BEGIN
26700                             COMMENT OUTTEXT("VALEURS OBSERVEES NULLES");
26750                               GO TO  FIN
26800                           END
26850                       ELSE  DENOM := DENOM / NB  ;
26900
26950     COMMENT   calcul du NUM = 1/nb-k somme de i jusque nb-k
27000                 (Xi-moy)(X[i+k]-moy) et test du rapport num/denom ;
```

```
27050          FOR K:=1 STEP 1 UNTIL NB DO BEGIN
27100           FOR I:=1 STEP 1 UNTIL NB-K DO
27150             NUM := NUM + (TAB[I]-MOY)*(TAB[I+K]-MOY) ;
27200           IF NB = K  THEN GO TO  CHOIX
27250                      ELSE NUM := NUM / (NB-K) ;
27300           IF ABS(NUM/DENOM) <= PARAM THEN GO TO CHOIX ;
27350           NUM := 0 ;
27400                                      END ;
27450
27500   COMMENT  choix de la  METHODE1 ou de la METHODE2
27550             pour le calcul des intervalles de confiance ;
27600   CHOIX :  IF  NB//K  <=  4  THEN  GO  TO  METHODE1 ;
27650
27700   COMMENT   ********
27750             METHODE2
27770             ******** ;
27790
27800   COMMENT       calcul des moyennes par bloc  ;
27850           FOR J:=1 STEP K UNTIL NB  DO
27900           BEGIN
27950             FOR I:=J STEP 1 UNTIL K+J-1  DO
28000             BEGIN
28050               SOMME := SOMME + TAB[I] ;
28100               IF I = NB  THEN  BEGIN
28150                                  K := NB-J+1 ;
28200                                  GO TO OUT
28250                                END ;
28300    OUT :      END ;
28350              NB_BLOCS := NB_BLOCS + 1 ;
28400              TAB[NB_BLOCS] := SOMME / K ;
28450              SOMME := 0 ;
28500           END  ;
28550
28600   COMMENT   calcul de la variance = 1 / nb_blocs -1
28650             somme i=1,nb_blocs Xj**2   -
28700             1/nb_blocs (somme i=1,nb_blocs Xj)**2  ;
28720             MOY := 0 ;
28740
28750           FOR I:=1 STEP 1 UNTIL NB_BLOCS  DO
28800           BEGIN
28850             SOMME := SOMME +  TAB[I]**2  ;
28900             VARIANCE := VARIANCE + TAB[I];
28950              MOY := MOY + TAB[I] ;
29000           END  ;
29050                     VARIANCE := (SOMME-(VARIANCE**2)/NB_BLOCS)/
29100                                 (NB_BLOCS - 1) ;
29120           ECART_TYPE := VARIANCE ** .5 ;
29135
29150   COMMENT   calcul de la moyenne des moyennes ;
29200                MOY := MOY / NB_BLOCS ;
29320
29340
29350   COMMENT  choix de la solution BIENAYME ou NORMALE
29400            pour le calcul des intervalles de confiance (methode2) ;
29425
29450      IF NB_BLOCS  >  30   THEN  BEGIN
29452                                   METHODE := 4 ;
29454                                   SOMME := 1.96 * ECART_TYPE ;
29456                                 END
29458                           ELSE BEGIN
```

```
29460                                   METHODE := 3 ;
29462                                   SOMME := 4.47 * ECART_TYPE ;
29464                                END ;
29466          GO TO FINAL ;
29600
29650     COMMENT    ********
29700                METHODE1
29725                ******** ;
29750
29800     METHODE1 :
29850                COMMENT   calcul de la somme des carres des observations ;
29900       SOMME := 0 ;
29950       FOR I := 1 STEP 1 UNTIL NB  DO  SOMME:=SOMME+TAB[I]**2 ;
30000       VARIANCE := (SOMME/NB) - (MOY**2) ;
30050       ECART_TYPE := VARIANCE ** .5 ;
30100
30150     COMMENT  choix de la solution BIENAYME ou NORMALE pour le calcul
30200              d'intervalles de confiance (Methode1)  ;
30250       IF  NB  >  30
30300       THEN  BEGIN
30350               METHODE := 2 ;
30400               SOMME := (1.96*ECART_TYPE) / (NB**.5) ;
30450             END
30500       ELSE  BEGIN
30550               METHODE := 1 ;
30600               SOMME := (4.47*ECART_TYPE) / (NB**.5) ;
30650             END ;
30700
30750
30800     FINAL : INTERMIN := MOY - SOMME ;
30850             INTERMAX := MOY + SOMME ;
30900
30950     COMMENT   tester la condition sur l'intervalle de confiance ;
31000             IF  ARRET_SIMULATION  THEN  BEGIN
31050
31100     COMMENT    edition du resultat :
31150                 si  METHODE =  1   BIENAYME (Methode1)
31200                                2   NORMALE  (Methode1)
31250                                3   BIENAYME (Methode2)
31300                                4   NORMALE  (Methode2)  ;
31350       SKIP(0) ; SETPOS(12) ;
31400       OUTINT(NB,4) ;
31450       IF METHODE = 3  OR  METHODE = 4  THEN  BEGIN
31500                    SETPOS(24) ; OUTINT(NB_BLOCS,4) ;
31550                    SETPOS(38) ; OUTTEXT("2") ;
31600                                              END
31650                                         ELSE  BEGIN
31700                    SETPOS(38) ; OUTTEXT("1") ;
31750                                              END ;
31800       IF METHODE = 1  OR  METHODE = 3  THEN  SETPOS(44)
31850                                         ELSE  SETPOS(55) ;
31900       OUTFIX(SOMME,3,10) ;
31950
32000
32050                    IF SOMME >= LONG_ADM
32100                    THEN BEGIN
32150     SKIP(2);
32200      OUTTEXT("ECART TYPE          :  "); OUTFIX(ECART_TYPE,3,15);SKIP(0);
32250      OUTTEXT("LONGUEUR DEMANDEE   :  "); OUTFIX(LONG_ADM,3,15); SKIP(0);
32300      OUTTEXT("LONGUEUR REELLE     :  "); OUTFIX(SOMME,3,15) ; SKIP(5) ;
```

```
32350
32400
32450                   CLEAR_SQS ;
32500
32550                                   ARRET_SIMULATION := FALSE ;
32600                                   REACTIVATE  MAIN ;
32650
32700
32750                   END ;
32800                                       END ;
32850
32900   FIN : END  PROCEDURE INTERVALLE  ;
32950
33000
33050   COMMENT   ************************************
33100      WAIT_UNTIL PARAPHENALIA
33150   ************************************;
33200
33250    REF  (HEAD) WAITQ ;
33300    REF  (WAIT_MONITEUR) WAIT_MONITOR ;
33350    BOOLEAN  WAIT_ACTION ;
33400
33450   PROCESS  CLASS  WAIT_MONITEUR ;
33500    WHILE   TRUE   DO
33550    BEGIN
33600      WAIT_ACTION :=  FALSE ;
33650      RESUME (WAITQ . FIRST);
33700       WHILE   NOT WAIT_ACTION  DO
33750       BEGIN
33800          REACTIVATE  CURRENT  AFTER  NEXTEV;
33850         RESUME (WAITQ . FIRST);
33900       END ;
33950       IF  WAITQ . EMPTY  THEN  PASSIVATE;
34000    END ;
34050
34100
34150    COMMENT
34200   ******************************
34250     PROCEDURES UTILITAIRES
34300   ******************************;
34350
34400    PROCEDURE  SKIP (N) ;
34450    INTEGER  N ;
34500    BEGIN
34550      OUTIMAGE ;
34600       IF  N > 0  THEN  EJECT (LINE + N)
34650      END  SKIP ;
34700
34750    REAL   PROCEDURE  GPSSS_TIME ;
34800      GPSSS_TIME := TIME - SIMULATION_START_TIME ;
34850
34900    PROCEDURE  CLEAR_SQS ;
34950    WHILE  CURRENT.NEXTEV =/=  NONE   DO  CANCEL(CURRENT.NEXTEV) ;
35000
35050    PROCEDURE  RESTART ;
35100    BEGIN
35150       IF  CURRENT =/= MAIN  THEN  ERREUR(9, NONE )
35200       ELSE
35250          BEGIN
35300            ERROR_REPORT ;
```

```
35350             STANDARD_REPORT ;
35400             PASSE := PASSE + 1 ;
35450           COMMENT  CANCEL ALL RESOURCES ;
35500             FACILITYQ.CLEAR ;
35550             REGIONQ.CLEAR ;
35600             STORAGEQ.CLEAR ;
35650             TABLEQ.CLEAR ;
35700           COMMENT  CANCEL ALL EVENTS ;
35750             CLEAR_SQS ;
35800             SIMULATION_START_TIME := TIME ;
35850             TRANS_ID := 0 ;
35900             U := 987654321
35950           END
36000       END  RESTART ;
36050
36100       PROCEDURE  RESET ;
36150       BEGIN
36200          REF  (ENTITY) P ;
36250         TIME_ORIGIN := TIME ;
36300          FOR  P :- FACILITYQ.FIRST, REGIONQ.FIRST,
36350                    STORAGEQ.FIRST, TABLEQ.FIRST  DO
36400             WHILE  P =/=  NONE   DO
36450               BEGIN
36500                 INSPECT  P
36550                   WHEN  FACILITY  DO
36600                     BEGIN
36650                       BOOLEAN  SBUS ;
36700                       SBUS := OCCUPIER =/=  NONE  ;
36750                       ENTRIES :=  IF  SBUS  THEN  1  ELSE  0 ;
36800                       BUSY_TIME := 0 ;
36850                       TLAST := TIME
36900                     END
36950                   WHEN  STORAGE  DO
37000                     BEGIN
37050                       ENTRIES := 0 ;
37100                       INTGRL := 0 ;
37150                       UNIT_ENTRIES := CONTENTS ;
37200                       TLAST := TIME ;
37250                       MAX := CONTENTS
37300                     END
37350                   WHEN  REGION  DO
37400                     BEGIN
37450                       ZERO_ENTRIES := 0 ;
37500                       INTGRL := 0 ;
37550                       MAX := ENTRIES := CONTENTS
37600                     END
37650                   WHEN  TABLE  DO
37700                     BEGIN
37750                       INTEGER  I ;
37800                       ENTRIES := 0 ;
37850                       MIN := MAX := SUM := 0 ;
37900                        FOR  I := -1  STEP  1  UNTIL  N  DO  A[I] := 0
37950                     END  ;
38000               P :- P.SUC
38050             END
38100       END  ;
38150
38200
38250       COMMENT ****************************************
38300             ENQUIRY PROCEDURES
```

```
38350   ***********************************************;
38400
38450    INTEGER   PROCEDURE  CONTENTS_FACILITY (F) ;  REF  (FACILITY) F ;
38500    INSPECT  F  DO
38550       IF  OCCUPIER =/=  NONE   THEN  CONTENTS_FACILITY := 1 ;
38600
38650    INTEGER   PROCEDURE  WAITING_FACILITY (F) ;  REF  (FACILITY) F ;
38700    INSPECT  F  DO
38750      WAITING_FACILITY := INQ.CARDINAL ;
38800
38850    INTEGER   PROCEDURE  CONTENTS_STORAGE (S) ;  REF  (STORAGE) S ;
38900    INSPECT  S  DO
38950      CONTENTS_STORAGE := CONTENTS ;
39000
39050    INTEGER   PROCEDURE  WAITING_STORAGE (S) ;  REF  (STORAGE) S ;
39100    INSPECT  S  DO
39150      WAITING_STORAGE := INQ.CARDINAL ;
39200
39250    INTEGER   PROCEDURE  CONTENTS_REGION (R) ;  REF  (REGION) R ;
39300    INSPECT  R  DO
39350      CONTENTS_REGION := CONTENTS ;
39400
39450    INTEGER   PROCEDURE  WAITING_REGION (R) ;  REF  (REGION) R ;
39500   WAITING_REGION := 0 ;
39550
39600    INTEGER   PROCEDURE  CONTENTS_GROUP (G) ;  REF  (GROUP) G ;
39650    INSPECT  G  DO
39700      CONTENTS_GROUP := N ;
39750
39800    INTEGER   PROCEDURE  WAITING_GROUP (G) ;  REF  (GROUP) G ;
39850   WAITING_GROUP := 0 ;
39900
39950    INTEGER   PROCEDURE  CONTENTS_TABLE (T) ;  REF  (TABLE) T ;
40000    INSPECT  T  DO
40050      CONTENTS_TABLE := ENTRIES ;
40100
40150    INTEGER   PROCEDURE  WAITING_TABLE (T) ;  REF  (TABLE) T ;
40200   WAITING_TABLE := 0 ;
40250
40300    COMMENT ************************************************
40350
40400            REPORT  GENERATOR
40450
40500   ************************************************************;
40550
40600    PROCEDURE  STANDARD_REPORT ;
40650    BEGIN
40700
40750       REF  (ENTITY) P ;
40800       REAL DTIME ;
40850
40900
40950
41000       PROCEDURE  FACILITIES_REPORT ;
41050       BEGIN
41100          BOOLEAN  SBUS ;
41150          IF  FACILITYQ.CARDINAL + 5 + LINE > 60  THEN  EJECT(1) ;
41200         OUTTEXT("* FACILITIES *") ; SKIP(0) ;
41250         OUTTEXT("***************") ; SKIP(1) ;
41300         SETPOS(25) ;
```

```
41350          OUTTEXT ("AVERAGE                        AVERAGE") ;
41400          SKIP(0) ;
41450          OUTTEXT ("                            UTILISATION") ;
41500          OUTTEXT ("     ENTRIES      TRANSIT TIME      STATUS") ;
41550          SKIP(1) ;
41600           FOR  P :- FACILITYQ.FIRST, P.SUC  WHILE  P=/= NONE   DO
41650           INSPECT  P  WHEN  FACILITY  DO
41700              BEGIN
41750                SBUS := OCCUPIER =/=  NONE  ;
41800                 IF  SBUS  THEN  BUSY_TIME:=BUSY_TIME+TIME-TLAST ;
41850                OUTTEXT (IDENT) ; SETPOS (18) ;
41900                OUTFIX(BUSY_TIME/DTIME,2,14) ;
41950                OUTINT(ENTRIES,14) ;
42000                 IF  ENTRIES  <=  0  THEN
42050                   OUTFIX(BUSY_TIME/ENTRIES,2,15) ;
42100                SETPOS(69) ;
42150                 IF  SBUS  THEN  OUTTEXT("BUSY")
42200                 ELSE  OUTTEXT("FREE") ;
42250                 SKIP(0) ;
42300              END ;
42350
42400          SKIP(1) ;
42450          OUTTEXT("FILE D'ATTENTE : FACILITIES") ; SKIP(0) ;
42500          OUTTEXT("--------------------------") ; SKIP(0) ;
42550          IMPRESSION ;
42600          FOR P :- FACILITYQ.FIRST, P.SUC  WHILE P=/=NONE  DO
42650          INSPECT P WHEN FACILITY  DO
42700          BEGIN
42750             SKIP(0) ;
42800             OUTTEXT(IDENT) ;      SETPOS(18) ;
42850             OUTINT(LISTE.CARDINAL,15) ;
42900              ACTIVATE NEW INTERVALLE(LISTE,LISTE.CARDINAL) ;
42950             SETPOS (33) ;  OUTFIX(MOY,3,15) ;
43000             SETPOS (48) ;  OUTFIX(INTERMIN,3,10) ;
43050             SETPOS (58) ;  OUTFIX(INTERMAX,3,10) ;
43100
43150              ACTIVATE NEW INTERVALLE (LISTE2,LISTE2.CARDINAL) ;
43200             SETPOS (68) ;  OUTFIX(MOY,3,15) ;
43250             SETPOS (83) ;  OUTFIX(INTERMIN,3,10) ;
43300             SETPOS (93) ;  OUTFIX(INTERMAX,3,10) ;
43350          END ;
43400        END  FACILITIES_REPORT  ;
43450
43500       PROCEDURE  STORAGES_REPORT ;
43550       BEGIN
43600          IF  STORAGEQ.CARDINAL + 5 + LINE > 60  THEN  EJECT(1) ;
43650          OUTTEXT("* STORAGES *") ; SKIP(0) ;
43700          OUTTEXT("************") ; SKIP(1) ;
43750          SETPOS (36) ;
43800          OUTTEXT ("AVG.        AVG.                        THRU TIME") ;
43850          OUTTEXT ("     CONTENTS") ;
43900          SKIP(0) ;
43950          SETPOS (21) ;
44000          OUTTEXT ("CAPACITY     CONTENTS     UTIL.") ;
44050          OUTTEXT ("     ENTRIES       / UNIT     MAX     NOW") ;
44100          SKIP (1) ;
44150           FOR  P :- STORAGEQ.FIRST,P.SUC  WHILE  P =/=  NONE   DO
44200           INSPECT  P  WHEN  STORAGE  DO
44250              BEGIN
44300                ACCUM(INTGRL.TLAST.CONTENTS,0) ;
```

```
44350              OUTTEXT (IDENT) ; SETPOS (16) ;
44400              OUTINT(CAPACITY,11) ;
44450              OUTFIX(INTGRL/DTIME,2,14) ;
44500               IF  CAPACITY = 0  THEN  SETPOS(52)
44550               ELSE  OUTFIX(INTGRL / (DTIME * CAPACITY),4,11) ;
44600              OUTINT(ENTRIES,10) ;
44650               IF  UNIT_ENTRIES = 0  THEN  SETPOS(75)
44700               ELSE  OUTFIX (INTGRL / UNIT_ENTRIES,2,13) ;
44750              OUTINT(MAX,9) ;
44800              OUTINT(CONTENTS,7) ;
44850              SKIP(0) ;
44900            END ;
44950
45000          SKIP(1) ;
45050          OUTTEXT("FILE D'ATTENTE : STORAGES") ; SKIP(0) ;
45100          OUTTEXT("-------------------------") ; SKIP(0) ;
45150        IMPRESSION ;
45200        FOR P :- STORAGEQ.FIRST, P.SUC  WHILE P=/=NONE  DO
45250        INSPECT P WHEN STORAGE  DO
45300          BEGIN
45350            SKIP(0) ;
45400            OUTTEXT(IDENT) ;       SETPOS(18) ;
45450            OUTINT(LISTE.CARDINAL,15) ;
45500             ACTIVATE NEW INTERVALLE(LISTE,LISTE.CARDINAL) ;
45550            SETPOS (33) ;  OUTFIX(MOY,3,15) ;
45600            SETPOS (48) ;  OUTFIX(INTERMIN,3,10) ;
45650            SETPOS (58) ;  OUTFIX(INTERMAX,3,10) ;
45700
45750             ACTIVATE NEW INTERVALLE (LISTE2,LISTE2.CARDINAL) ;
45800            SETPOS (68) ;  OUTFIX(MOY,3,15) ;
45850            SETPOS (83) ;  OUTFIX(INTERMIN,3,10) ;
45900            SETPOS (93) ;  OUTFIX(INTERMAX,3,10) ;
45950          END ;
46000            END  STORAGES_REPORT  ;
46050
46100        PROCEDURE  IMPRESSION  ;
46150          BEGIN
46200          SETPOS (27) ;
46250          OUTTEXT("NOMBRE          NOMBRE       INTERVALLE     ") ;
46300          OUTTEXT("       TEMPS        INTERVALLE") ; SKIP(0) ;
46350          SETPOS(27) ;
46400          OUTTEXT("D'OBS.          MOYEN         MIN    MAX     ") ;
46450          OUTTEXT("       MOYEN        MIN     MAX") ; SKIP(1) ;
46500          END  PROCEDURE  IMPRESSION ;
46550
46600
46650        PROCEDURE  REGIONS_REPORT ;
46700        BEGIN
46750          IF  REGIONQ.CARDINAL + 5 + LINE > 50  THEN  EJECT(1) ;
46800          OUTTEXT ("* REGIONS *") ; SKIP(0) ;
46850          OUTTEXT ("**********") ; SKIP(1) ;
46900          SETPOS (52) ;
46950          OUTTEXT ("MAX.          CURRENT          AVG.             ") ;
47000          OUTTEXT ("AVG. TIME/TRANS") ;
47050          SKIP(0) ;
47100          OUTTEXT ("                    ENTRIES     ZERO_ENTRIES") ;
47150          OUTTEXT ("     CONTENTS     ") ;
47200          OUTTEXT ("CONTENTS     CONTENTS          ALL       NON ZERO") ;
47250          SKIP (1) ;
47300           FOR  P :- REGIONQ.FIRST,P.SUC  WHILE  P =/=  NONE   DO
```

```
47350           INSPECT  P  WHEN  REGION  DO
47400              BEGIN
47450                ACCUM (INTGRL, TLAST, CONTENTS, 0) ;
47500                OUTTEXT (IDENT) ; SETPOS (16) ;
47550                OUTINT (ENTRIES, 10) ;
47600                OUTINT (ZERO_ENTRIES, 15) ;
47650                OUTINT (MAX, 15) ;
47700                OUTINT (CONTENTS, 13) ;
47750                OUTFIX (INTGRL / DTIME, 2, 13) ;
47800                 IF  ENTRIES  \=  0  THEN
47850                   BEGIN
47900                     OUTFIX (INTGRL / ENTRIES, 2, 15) ;
47950                      IF  ENTRIES  \=  ZERO_ENTRIES  THEN
48000                        OUTFIX (INTGRL / (ENTRIES - ZERO_ENTRIES), 2, 15)
48050                   END  ;
48100                SKIP (0) ;
48150              END
48200        END  REGIONS_REPORT ;
48250
48300        PROCEDURE  TABLES_REPORT ;
48350        BEGIN
48400           BOOLEAN  DETAIL ;
48450           INTEGER  CPL, LPSS, TL, I, J, K ;
48500
48550        PROCEDURE  HISTO (T) ;  REF  (TABLE) T ;
48600        BEGIN
48650           TEXT  STARS ;
48700           REAL  SCALELEVEL, LINESTEP ;
48750           INTEGER  FREQMAX, CHAMP, NBCAR, SCALE, SCALESTEP ;
48800
48850           INSPECT  T  DO
48900              BEGIN
48950                COMMENT  TROUVER LA FREQUENCE MAXIMUM ;
49000                  FREQMAX := A[-1] ;
49050                  FOR  I := 0  STEP  1  UNTIL  N  DO
49100                     IF  FREQMAX < A[I]  THEN  FREQMAX := A[I] ;
49150                COMMENT  TROUVER L ECHELLE (MIN = 10) ;
49200                  SCALE := 10 ;
49250                  NBCAR := 2 ;
49300                  IF  FREQMAX > 10  THEN
49350                     BEGIN
49400                       WHILE  FREQMAX > SCALE  DO
49450                          BEGIN
49500                            SCALE := SCALE * 10 ;
49550                            NBCAR := NBCAR + 1
49600                          END  ;
49650                       IF  SCALE > FREQMAX  THEN
49700                          BEGIN
49750                            NBCAR := NBCAR - 1 ;
49800                             IF  SCALE  //  5 > FREQMAX  THEN
49850                               SCALE := SCALE  //  5
49900                             ELSE   IF  SCALE  //  2 > FREQMAX  THEN
49950                               SCALE := SCALE  //  2 ;
50000                          END
50050                     END  ;
50100                  SCALESTEP := SCALE  //  10 ;
50150                  LINESTEP := SCALESTEP / LPSS ;
50200                COMMENT  TROUVER LA LARGEUR D UNE COLONNE ;
50250                  CHAMP := (CPL-10)  //  (N+2) ;
50300                  IF  CHAMP > 20  THEN  CHAMP := 20 ;
```

```
50350                 STARS :- COPY (" ********************") ;
50400                 STARS :- STARS.SUB(1,CHAMP) ;
50450               COMMENT  IMPRIMER LES TETES DE COLONNES SI POSSIBLE ;
50500                 OUTTEXT ("ENTRIES  :") ;
50550                  IF  NBCAR > CHAMP-1  THEN  DETAIL :=  TRUE
50600                  ELSE   FOR  I := -1  STEP  1  UNTIL  N  DO
50650                            OUTINT (A[I], CHAMP) ;
50700                 OUTIMAGE ;
50750                  FOR  I := 1  STEP  1  UNTIL  CHAMP*(N+2)+10  DO
50800                       OUTCHAR ('.') ;
50850                 OUTIMAGE ;
50900               COMMENT  TRACER LES COLONNES ;
50950                  FOR  I := 1  STEP  1  UNTIL  10  DO
51000                     BEGIN
51050                       OUTINT (SCALE, 8) ;
51100                       SCALELEVEL := SCALE + LINESTEP / 2 ;
51150                       SCALE := SCALE - SCALESTEP ;
51200                        FOR  J := 1  STEP  1  UNTIL  LPSS  DO
51250                           BEGIN
51300                             SETPOS (9) ;
51350                             OUTTEXT (" :") ;
51400                             SCALELEVEL := SCALELEVEL - LINESTEP ;
51450                              FOR  K := -1  STEP  1  UNTIL  N  DO
51500                                 IF  A[K] < SCALELEVEL
51550                                   THEN  SETPOS (POS+CHAMP)
51600                                   ELSE  OUTTEXT (STARS) ;
51650                             OUTIMAGE
51700                           END
51750                     END  ;
51800                  FOR  I := 1  STEP  1  UNTIL  CHAMP*(N+2)+10  DO
51850                    OUTCHAR ('.') ;
51900                 OUTIMAGE ;
51950               COMMENT  IDENTIFIER LES INTERVALLES ;
52000                 OUTTEXT ("INTERVAL :") ;
52050                  FOR  I := 0  STEP  1  UNTIL  N+1  DO  OUTINT (I,CHAMP) ;
52100                 SKIP (2) ;
52150          END
52200       END     HISTO ;
52250
52300         CPL := 132 ;          COMMENT  CHARACTER PER LINE ;
52350         LPSS := 4 ;           COMMENT  LINES PER SCALE S STEP ;
52400         TL := LPSS * 10 ;     COMMENT  TOTAL LINES FOR THE HISTOGRAM ;
52450
52500          FOR  P :- TABLEQ.FIRST, P.SUC  WHILE  P =/=  NONE   DO
52550          INSPECT  P  QUA  TABLE DO
52600          BEGIN
52650            EJECT (1) ;
52700            OUTTEXT ("* TABLE *   ") ; OUTTEXT (IDENT) ; OUTIMAGE ;
52750            OUTTEXT ("*********") ; SKIP (2) ;
52800             IF  (N+2) > ((CPL-10)  //  3)  THEN  DETAIL :=  TRUE
52850             ELSE  HISTO (P) ;
52900
52950          COMMENT  DECRIRE LES INTERVALLES ;
53000            OUTTEXT(" INTERVAL WIDTH   = ") ; OUTFIX(INTER,3,15) ;
53050            OUTIMAGE ;
53100            OUTTEXT(" INTERVAL NO.  1  = ") ; OUTFIX(LOWER,3,15) ;
53150            OUTTEXT("  TO ") ; OUTFIX(LOWER+INTER,3,15) ;
53200            OUTIMAGE ;
53250            OUTTEXT("INTERVAL NO. ") ; OUTINT(N,2) ; OUTTEXT("  =") ;
53300            OUTFIX(LOWER+(N-1)*INTER,3,15) ; OUTTEXT("  TO ") ;
```

```
53350            OUTFIX(LOWER+N*INTER,3,15) ; OUTIMAGE ;
53400             SKIP (2) ;
53450
53500             OUTTEXT(" TOTAL ENTRIES = ") ; OUTINT(ENTRIES,15) ;
53550             OUTIMAGE ;
53600             OUTTEXT(" AVERAGE       = ") ; OUTFIX(SUM/ENTRIES,3,15) ;
53650             OUTIMAGE ;
53700             OUTTEXT(" MIN VALUE     = ") ; OUTFIX(MIN,3,15) ;
53750             OUTIMAGE ;
53800             OUTTEXT(" MAX VALUE     = ") ; OUTFIX(MAX,3,15) ;
53850             OUTIMAGE ;
53900
53950              IF  DETAIL  THEN
54000                 BEGIN
54050                   EJECT(1) ;
54100                   OUTTEXT(" * TABLE *     ") ; OUTTEXT(IDENT) ;
54150                   OUTTEXT("    NB OF ENTRIES FOR EACH INTERVAL ") ;
54200                   SKIP (3) ;
54250                   K := CPL  //  25 ;
54300                   J := 0 ;
54350                    FOR  I := -1  STEP  1  UNTIL  N  DO
54400                       BEGIN
54450                         OUTTEXT ("INT. ") ;
54500                         OUTINT (I+1, 2) ;
54550                         OUTTEXT (" =") ;
54600                         OUTINT (A[I], 11) ;
54650                         SETPOS(POS+5) ;
54700                         J := J + 1 ;
54750                          IF  J = K  THEN
54800                             BEGIN  OUTIMAGE ; J := 0  END
54850                       END
54900                 END
54950          END
55000       END  TABLE REPORT ;
55050
55100       COMMENT   DEBUT, PROCEDURE STANDARD_REPORT ;
55150
55200      DTIME := TIME - TIME_ORIGIN ;
```

```
55250       EJECT(1) ;
55300       CLEAR_SQS ;  CURRENT.OUT ;
55350       SKIP(3)  ;
55400       OUTTEXT("****************************") ;
55450       OUTTEXT("   -VERSION 5.0-") ; OUTIMAGE ;
55500       OUTTEXT("***   MONTREAL  GPSSS   ***") ; SKIP(0) ;
55550       OUTTEXT("***  SIMULATION REPORT  ***") ; SKIP(0) ;
55600       OUTTEXT("****************************") ; SKIP(2) ;
55650       OUTTEXT("PASSE =") ;
55700       OUTINT(PASSE,3) ;
55750       SKIP(1) ;
55800       OUTTEXT("START TIME=") ;
55850       OUTFIX(TIME_ORIGIN - SIMULATION_START_TIME,2,10) ;
55900       SKIP(0) ;
55950       OUTTEXT("END   TIME=") ;
56000       OUTFIX(GPSSS_TIME,2,10) ;
56050        IF  DTIME = 0  THEN
56100           BEGIN
56150             SKIP(2) ;
56200             OUTTEXT("ELAPSED TIME = 0 - REPORT SKIPPED")
56250           END
56300        ELSE
56350           BEGIN
56400             SKIP(2) ;
56450              IF  FACILITYQ.EMPTY  THEN  OUTTEXT ("* NO FACILITIES *")
56500              ELSE  FACILITIES_REPORT ;
56550             SKIP(2) ;
56600              IF  STORAGEQ.EMPTY  THEN  OUTTEXT ("* NO STORAGES *")
56650              ELSE  STORAGES_REPORT ;
56700             SKIP(2) ;
```

```
55750             IF  REGIONQ.EMPTY  THEN  OUTTEXT ("* NO REGIONS *")
56800             ELSE  REGIONS_REPORT ;
56850            SKIP(2) ;
56900             IF  TABLEQ.EMPTY  THEN  OUTTEXT ("* NO TABLES *")
56950             ELSE  TABLES_REPORT
57000          END  ;
57050       OUTIMAGE ;
57100       RESET
57150
57200
57250    END  STANDARD_REPORT ;
57300
57350    COMMENT  ********************************************
57400             ***   TRACE   PROCEDURE                    ***
57450             ********************************************  ;
57500
57550       REF (PT_TRACE) TR ;
57600       REF (TRANS) TQ    ;
57650       BOOLEAN  TRACE_GEN ;
57700       BOOLEAN  INIT_TRACE ;
57750       INTEGER   MAX_TRACE ;
57800
57850    PROCEDURE  TRACE(N) ;  INTEGER N ;
57900    BEGIN
57950       IF NOT INIT_TRACE  THEN  BEGIN
58000                                  TRACE_GEN := INIT_TRACE := TRUE ;
58050                                  MAX_TRACE := N ;
58100                                END
58150                           ELSE  BEGIN
58200          SKIP(3) ;  OUTTEXT("***************************") ;
58250          SKIP(0) ;  OUTTEXT("***   TRACE  REPORT   ***") ;
58300          SKIP(0) ;  OUTTEXT("***************************") ; SKIP(2) ;
58350
58400          FOR TR:-TRANSACTIONQ.FIRST,TR.SUC WHILE TR=/=NONE  DO
58450            INSPECT  TR  WHEN  PT_TRACE  DO
58500          BEGIN
58550            IF  NOT FILE_CREATED  THEN  GO TO  FIN  ;
58600            SKIP(1);
58650            OUTTEXT("TRANSACTION   ") ;  OUTINT(IDENTIFICATEUR,4) ;
58700            IF  MARQUE  \=  CHAR(0)  THEN  BEGIN
58750                                     OUTTEXT("  TYPE IS   ");
58800                                     OUTCHAR(MARQUE)
58850                                       END  ;
58900            SKIP(0) ;  OUTTEXT("------------------") ;
58950
59000
59050
59100            FOR TQ:-FILE_TRACE.FIRST,TQ.SUC WHILE TQ=/=NONE  DO
59150              INSPECT TQ WHEN TRANS  DO
59200            BEGIN
59250              SKIP(0) ;
59300              IF OPERATION = 0  THEN  OUTTEXT("ENTERED   ")
59350                                ELSE  OUTTEXT("LEAVED    ");
59400              OUTTEXT(NOM) ;  SETPOS(25) ;  OUTTEXT("   AT TIME    ") ;
59450              OUTINT(TEMPS_TRACE,5) ;
59500            END  ;
59550          END  ;
59600                                END ;
59650    FIN :
59700
```

```
59750      END   PROCEDURE  TRACE  ;
59800
59850
59900
59950    COMMENT   ***************************************
60000
60050                PROCEDURE  OBSERVATION_AUTOMATIQUE
60100
60150                *************************************** ;
60200
60250
60300    BOOLEAN  COLLECT_AUTOM ;
60350
60400    PROCEDURE OBSERVATION_AUTOMATIQUE (FREQUENCE) ;
60450                INTEGER  FREQUENCE ;
60500     BEGIN
60550       COLLECT_AUTOM  :=  TRUE  ;
60600       TIME_BTW_OBS := FREQUENCE ;
60650       ACTIVATE  NEW  OBSERVATION  DELAY  0  ;
60700     END ;
60750
60800
60850    COMMENT   *****************************************
60900
60950                PROCEDURE  INTER_CONFIANCE
61000
61050                ***************************************** ;
61100
61150    INTEGER  I_C_TYP ;
61200    REAL       PARAM ,  LONG_ADM ;
61250    REF(ENTITY) I_C_NOM ;
61300    BOOLEAN  ARRET_SIMULATION  ;
61350
61400
61450    PROCEDURE INTER_CONFIANCE (ENTITY_NOM,RAPPORT,LONGUEUR,TYPE);
61500          NAME ENTITY_NOM ;
61550          REF(ENTITY) ENTITY_NOM ; REAL RAPPORT ; REAL LONGUEUR ; INTEGER TYPE ;
61600       BEGIN
61650          I_C_NOM    :=  ENTITY_NOM ;
61700          PARAM      :=  RAPPORT    ;
61750          LONG_ADM   :=  LONGUEUR   ;
61800          I_C_TYP    :=  TYPE       ;
61850          ARRET_SIMULATION := TRUE ;
61900
61950    SKIP(3); OUTTEXT("*************************************");
62000    SKIP(0); OUTTEXT("***   INTERVALLE  DE  CONFIANCE ***");
62050    SKIP(0); OUTTEXT("*************************************");
62100    SKIP(2);
62150
62200    SKIP(0); OUTTEXT("TYPE D'OBSERVATIONS :  ");
62250    IF I_C_TYP=0 THEN BEGIN
62300             OUTTEXT("NOMBRE MOYEN DE TRANSACTIONS EN ATTENTE");
62350                END
62400                ELSE IF I_C_TYP=1 THEN BEGIN
62450             OUTTEXT("TEMPS MOYEN D'ATTENTE DES TRANSACTIONS");
62500                                    END
62550                                    ELSE OUTTEXT("PAS DEFINI");
62600     SKIP(0); OUTTEXT("PARAMETRE DEMANDE DE COVARIANCE  :  ");
62650     OUTFIX (PARAM,3,15);
62700     SKIP(0); OUTTEXT("LONGUEUR DEMANDEE DE L'INTERVALLE :  ");
```

```
62750     OUTFIX(LONG_ADM,3,15); SKIP(2);
62800
62850    SETPOS(52) ; OUTTEXT("LONGUEUR"); SKIP(0) ;
62900    SETPOS(10) ; OUTTEXT("NB D'OB.     NB BLOCS     METHODE") ;
62950    OUTTEXT("     BIENAYME     NORMALE") ; SKIP(0) ;
63000    SETPOS(10) ; outtext("--------     --------     -------") ;
63050    OUTTEXT("     --------     -------") ; SKIP(1) ;
63100
63150
63200     END ;
63250
63300     COMMENT  ******************************
63350
63400       ERROR    PACKAGE
63450
63500    ****************************************;
63550
63600     INTEGER  ERNUM ;
63650     INTEGER   ARRAY  ERTYPE[1:15], ERTRAN[1:15] ;
63700     REF  (ENTITY)  ARRAY  ERRES [1:15] ;
63750     REAL   ARRAY  ERTIME [1:15] ;
63800
63850     PROCEDURE  ERREUR (N,ERLOC) ;
63900     INTEGER  N ;
63950     REF  (ENTITY) ERLOC ;
64000     BEGIN
64050       ERNUM := ERNUM + 1 ;
64100       ERTIME [ERNUM] := GPSSS_TIME ;
64150        IF  N<2  OR  N>12  THEN  ERTYPE[ERNUM] := 1
64200        ELSE
64250           BEGIN
64300             ERTYPE[ERNUM] := N ;
64350             ERRES [ERNUM] :- ERLOC ;
64400              INSPECT  CURRENT  WHEN  TRANSACTION  DO
64450                ERTRAN [ERNUM] := ID
64500              OTHERWISE
64550                ERTRAN [ERNUM] := 0
64600           END  ;
64650        IF  ERNUM = 14  THEN
64700           BEGIN
64750             ERREUR(2, NONE ) ;
64800             CLEAR_SQS ;
64850             ERROR_REPORT ;
64900              IF  CURRENT =/= MAIN  THEN
64950                 BEGIN
65000                   ACTIVATE  MAIN  DELAY  0 ;
65050                   PASSIVATE
65100                 END
65150           END
65200     END  ERREUR ;
65250
65300     PROCEDURE  ERROR_REPORT ;
65350     BEGIN
65400        INTEGER  I ;
65450
65500        PROCEDURE  PHRASE1 (N) ;  INTEGER  N ;
65550        BEGIN
65600           IF  N=1  THEN  OUTTEXT("IN FACILITY")
65650           ELSE   IF  N=2  THEN  OUTTEXT("IN STORAGE")
65700           ELSE   IF  N=3  THEN  OUTTEXT("IN REGION")
```

```
65750           ELSE  OUTTEXT("IN XXX") ;
65800           SETPOS (POS + 1) ;
65850           OUTTEXT (ERRES[I].IDENT) ;
65900           OUTIMAGE
65950         END  ;
66000         PROCEDURE PHRASE2  ;
66050           BEGIN
66100           OUTTEXT("                TRANSACTION") ;
66150           OUTINT(ERTRAN[I],4)
66200         END  ;
66250
66300         IF  ERNUM  \=  0  THEN
66350         BEGIN
66400           SKIP(100) ;
66450           OUTTEXT("GPSSS ERROR_REPORT   - AT TIME=") ;
66500           OUTFIX(GPSSS_TIME,2,10) ;
66550           OUTIMAGE ;
66600           OUTTEXT("******************") ; SKIP(1) ;
66650           OUTTEXT("PASSE =") ;
66700           OUTINT(PASSE,3) ;
66750           SKIP(1) ;
66800           OUTTEXT("** TIME **") ; OUTIMAGE ;
66850            FOR  I:=1  STEP  1  UNTIL  ERNUM  DO
66900            BEGIN
66950               SWITCH  MSG := L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12 ;
67000              OUTIMAGE ;
67050              OUTFIX(ERTIME[I],2,10) ;
67100              OUTTEXT("  - ") ;
67150               GOTO  MSG[ERTYPE[I]];
67200           L1:OUTTEXT("SYSTEM ERROR") ;
67250               GOTO  OUT ;
67300           L2:OUTTEXT("TOO MANY ERRORS - SIMULATION TERMINATED");
67350               GOTO  OUT ;
67400           L3:PHRASE1 (1) ; PHRASE2 ;
67450              OUTTEXT(" TRIES TO LEAVE BEFORE ENTERING") ;
67500               GOTO  OUT ;
67550           L4:PHRASE1 (2) ; PHRASE2 ;
67600              OUTTEXT(" REQUIRES MORE THAN MAX CAPACITY") ;
67650               GOTO  OUT ;
67700           L5:PHRASE1 (2) ; PHRASE2 ;
67750              OUTTEXT(" CAUSES OVERFLOW WHEN LEAVING") ;
67800               GOTO  OUT ;
67850           L6:PHRASE1 (1) ; PHRASE2 ;
67900              OUTTEXT(" TRIES TO RE-ENTER") ;
67950               GOTO  OUT ;
68000           L7:PHRASE1 (3) ; PHRASE2 ;
68050              OUTTEXT(" TRIES TO LEAVE EMPTY REGION") ;
68100               GOTO  OUT ;
68150           L8:PHRASE2 ;
68200
68250
68300              OUTTEXT (" TRIES TO JOIN A NON-EXISTENT GROUP") ;
68350               GOTO  OUT ;
68400           L9:OUTTEXT("RESTART USED OUTSIDE MAIN PROGRAM");
68450              OUTTEXT(" - NO ACTION") ;
68500               GOTO  OUT ;
68550          L10:PHRASE2 ;
68600              OUTTEXT (" TRIES TO LEAVE NON-EXIXTENT FACILITY") ;
68650               GOTO  OUT ;
68700          L11:PHRASE2 ;
```

```
68750             OUTTEXT(" TRIES TO LEAVE NON-EXISTENT STORAGE") ;
68800              GOTO  OUT ;
68850          L12:PHRASE2 ;
68900              OUTTEXT (" TRIES TO LEAVE NON-EXISTENT REGION") ;
68950           OUT:
69000              OUTIMAGE ;
69050            END  ;
69100            ERNUM := 0
69150         END
69200      END   ERROR_REPORT ;
69250
69300      COMMENT ************************************
69350        INITIALISATION
69400     ******************************************;
69450
69500
69550        U :=987654321;
69600        PASSE := 1 ;
69650        PARAM :=   .2  ;
69700        FACILITYQ :-  NEW  HEAD ;
69750        REGIONQ   :-  NEW  HEAD ;
69800        STORAGEQ  :-  NEW  HEAD ;
69850        TRANSACTIONQ :-  NEW HEAD ;
69900        TABLEQ    :-  NEW  HEAD ;
69950        WAITQ     :-  NEW  HEAD ;
70000        WAIT_MONITOR :-  NEW  WAIT_MONITEUR ;
70050
70100         INNER  ;
70150
70200         IF TRACE_GEN  THEN  BEGIN
70250                              TRACE(0) ;  SKIP(2) ;
70300                              OUTTEXT("MAX_TRACE") ;
70350                              OUTINT(MAX_TRACE,5)  ;
70400                             END ;
70450        ERROR_REPORT ;
70500
70550        END  GPSSS  DEFINITION
```